НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

"На правах рукопису"
УДК 004.021

«До захисту допущено»
Завідувач кафедри
О.В. Коваль
(підпис) (ініціали, прізвище)
"___"_____2018р.

Магістерська дисертація

зі спеціальності - 122 Комп'ютерні науки та інформаційні технологій
за спеціалізацією - Геометричне моделювання в інформаційних системах
на тему: Моделювання функціональної верифікації інтерфейсу програмної системи

Виконав: студент 6 курсу, групи ТР – 71мп
Крамар Олена Валеріївна _____
(прізвище, ім'я, по батькові) (підпис)

Керівник к.т.н., доцент Коваль О. В. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент к.т.н., доцент Іванова Л. М. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2018

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти другий , магістерський

зі спеціальності - 122 Комп’ютерні науки та інформаційні технології
за спеціалізацією - Геометричне моделювання в інформаційних системах

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2018р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Крамар Олені Валеріївні
(прізвище, ім’я, по батькові)

1. Тема дисертації Моделювання функціональної верифікації інтерфейсу програмної системи
Науковий керівник к.т.н., доцент Коваль Олександр Васильович
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)
затверджені наказом по університету від ”05” 11 2018р. №4072-с
2. Строк подання студентом дисертації “12” грудня 2018 року
3. Об’єкт дослідження: методи та інструменти тестування програмного забезпечення складних програмних систем
4. Предмет дослідження: технології та засоби забезпечення верифікації інтерфейсів у системі слабо зв’язаних програмних компонентів
5. Перелік питань, які потрібно розробити:
1) проаналізувати сучасні архітектурні стилі;
2) дослідження ролі програмних інтерфейсів у мікросервісній архітектурі;
3) спроектувати абстракцію високого рівня для тестового фреймоворку;
4) дослідження особливостей налагодження мікросервісів.
6. Орієнтований перелік ілюстративного матеріалу: архітектура системи слабо зв’язаних компонентів, інтеграції з джерелами, схеми рівнів тестування

7. Дата видачі завдання ”30” вересня 2017р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Строки виконання етапів магістерської дисертації	Приміпка
1	Отримання завдання	30.09.17р.	
2	Збір інформації	02.10.17р. — 18.11.17р.	
3	Аналіз вимог завдання, вибір методів і засобів розв'язання поставленої задачі	19.11.17р. — 31.01.18р.	
4	Підготовка матеріалів магістерської роботи	05.02.18р. — 11.05.18р.	
5	Проміжний контроль підготовки	02.04.18р. — 06.04.18р.	
6	Написання основних розділів дисертації	12.09.18р. — 25.11.18р.	
7	Звіт з роботи над магістерською дисертацією	05.12.18р.	

Студент Крамар О. В.

(підпис) (прізвище та ініціали)

Керівник роботи Коваль О. В.

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Структура й обсяг дипломної роботи. Магістерська дисертація складається зі вступу, чотирьох розділів, висновку, переліку посилань з 18 найменувань, 1 додаток, і містить 23 рисунки, 11 таблиць. Повний обсяг магістерської дисертації складає 89 сторінок, з яких перелік посилань займає 2 сторінки, додатки – 2 сторінки.

Актуальність теми. Сучасне програмне забезпечення для його кращого використання та поширення будується таким чином, що інтерфейс взаємодії з системою представляється у формі API. API підлягають тестуванню. Але для програмних систем, що будуються за принципом модульності та слабкого зв'язування програмних модулів, процес тестування роботи системи дуже уповільнюється, адже щоб протестувати систему потрібно очікувати моменту завершення розробки всіх складових програмної системи. Тому необхідно запропонувати рішення, в якому тестування окремих частин системи можливе без готовності всіх її компонентів. Для вирішення цієї проблеми можливо застосувати підхід, що базується на контрактному тестуванні.

Мета дослідження полягає в створенні засобів та виявлені нових підходів для організації та проведення процесів тестування сучасного програмного забезпечення.

Об'єктом дослідження є методи та інструменти тестування програмного забезпечення складних програмних систем.

Предметом дослідження є технології та засоби забезпечення верифікації інтерфейсів у системі слабо зв'язаних програмних компонентів.

Методи дослідження.

– класифікація, моделювання (при формалізованому описі контрактного тестування, засобів побудови тестів з верифікації інтерфейсів);

– аналіз і проектування ПЗ (при розробці програмного модулю з функціональній верифікації);

– експеримент (при дослідженні доцільності використання запропонованого підходу та працездатності розроблених програмних засобів).

Наукова новизна одержаних результатів. Найбільш суттєвими науковими результатами магістерської дисертації є:

– запропоновано підхід до верифікації інтерфейсів у системі слабо зв’язаних програмних компонентів, що базується на контрактному тестуванні;

– набуло подальшого розвитку застосування методів та процесів автоматизованого тестування компонентів програмного забезпечення.

Практичне значення. Розроблений бібліотечний фреймворк для написання та виконання тестів з верифікації інтерфейсів у системі слабо зв’язаних програмних компонентів пропонується використовувати як допоміжний інструмент при розробці програмного забезпечення з мікросервісною архітектурою.

Ключові слова: МІКРОСЕРВІСНА АРХІТЕКТУРА, КОНТРАКТНЕ ТЕСТУВАННЯ, ТЕСТУВАННЯ СЛАБО ЗВ’ЯЗАНИХ ПРОГРАМНИХ СИСТЕМ, ТЕСТУВАННЯ ПРОГРАМНОГО ІНТЕРФЕЙСУ

Зміст

Вступ

Роль API (Application Programming Interface) в розробці та використанні вихідного коду програмного забезпечення важко недооцінити. Сучасні API часто приймають форму веб-сервісів, які надають користувачам (як людям, так і іншим веб-сервісам) деяку інформацію. Зазвичай ця процедура обміну інформацією і формат передачі даних структуровані, щоб обидві сторони знали, як взаємодіяти між собою.

Як любий вихідний код інтерфейси підлягають тестуванню. При тестуванні API потрібно враховувати, що API створюються переважно для інтеграції з іншими сервісами. І користуються ними не людина, а інші програмні системи. Тому потрібно також оцінювати API з позиції зручності його використання разом з іншими продуктами, з позиції легкої інтеграції. Кожен API повинен бути гнучким, мати зрозумілу і детальну документацію.

Але у складних програмних системах часто виникає ситуація, коли частина її компонентів не досяжна до використання, не працююча, або умовно працююча (знаходиться у стадії розробки, перероблюється, замінюється більш новою версією чи в ній знайдено помилки і вона тимчасово визнана неспроможною виконувати поставлені до неї завдання).

В таких умовах процес тестування роботи системи дуже уповільнюється, адже щоб протестувати систему на працездатність потрібно очікувати моменту завершення розробки всіх складових програмної системи. Тому потрібно шукати чи запропонувати рішення, коли тестування окремих частин системи можливе без повної готовності всіх її компонентів.

Постановка та вирішення такої задачі можлива лише до програмних систем, що мають не монолітну архітектуру, а будуються за принципом модульності та відокремлення та слабкого зв’язування програмних модулів. Це надає можливість абстрагуватись від залежних частин при тестуванні окремого компонента, але потребує формування та застосування нових підходів та засобів, що спрощують їх використання.

Це можливо за умови глибокого розуміння принципів проектування та роботи програмних систем із вказаною архітектурою, методів тестування та інструментів, що дозволяють застосувати дані методи.

Тому магістерська дисертація присвячена пошуку засобів та шляхів вирішення задачі верифікації інтерфейсів у системі слабо зв’язаних програмних компонентів.

1. Верифікація інтерфейсів у системі слабо зв'язаних програмних компонентів

1.1. Роль інтерфейсів у системі слабо зв'язаних програмних компонентів

У сучасному світі розробки програмного забезпечення доля коду пакетних програмних продуктів, що виконується через користувацький (людино-машинний) інтерфейс, стрімко зменшується. На зміну йому приходить код великих програмних систем, який предоставляється у використання не людиною, а в свою чергу кодом – іншим компонентом системи. Такі компоненти є достатньо незалежними, слабо зв'язаними. Для їх розробки вже створено та задокументовано багато проектних та архітектурних рішень. Правила взаємодії з таким кодом, тобто що і як викликати, які дані передати, на який формат результату очікувати – все це описується спеціальним узгодженням або інтерфейсом. Такого виду узгодження між програмними компонентами називаються API (Application Programming Interface) (рис 1.1).

Рисунок 1.1 - узгодження між програмними компонентами через API

Програмісту при написанні свого коду не потрібно мати у доступі вихідний код компонентів, що використовуються, достатньо мати знання як їх використовувати. У якості знань виступає саме API. У випадку веб-систем

основним форматом даних для обміну був HTML. Крім HTML наразі використовується низка сучасних протоколів та стандартів: SOAP, XML, JSON. Web-сервіси, отримавши розвиток у 2000 роках, орієнтувались на XML та SOAP. Більш сучасні REST-сервіси пропонують дані переважно у JSON-форматі. Однак якими би різними не були формати та типи даних для API веб-систем, вони повинні бути сумісними з протоколом HTML.

Для прикладу можна розглянути один з загально відомих веб-систем Github, він має свій API, яким можуть скористатися інші розробники. Те, як розробники користають його, залежить від можливостей, що надає API і від потреб розробників. API Github дозволяє, наприклад, отримувати інформацію про користувача, його аватар (зображення), читачів, репозиторії і багато іншої корисної інформації про контент та користувачів ресурсу.

Інший приклад, API Твіттера, декларує можливість цього сервісу видати інформацію про твіти користувача, його читачів і їх відомості та багато іншого. Це лише мала частина можливостей, які можливо застосувати, використовуючи API стороннього сервісу або створюючи свій власний.

Також на основі API будуються такі продукти, як карти 2GIS, різноманітні мобільні і десктопні клієнти для Twitter і Facebook. Всі їх функції стали можливими саме завдяки тому, що відповідні сервіси мають якісні і детально задокументовані API.

Існує багато сценаріїв, в яких постає потреба або доцільне рішення створити API для власного коду.

Найбільше поширеними є наступні:

- Мобільний додаток. Тому що безліч мобільних додатків для різних сервісів працюють при використанні API цих самих сервісів. Розробник додатку описав API, сформував певний навіть нескладний мобільний додаток, і клієнт зі смартфоном буде отримувати інформацію в свій пристрій саме через API. Це є зручним та сучасним технічним рішенням.

- Open course - код. Якщо у застосування склалася певна аудиторія, яка користується ним, э певний сенс та користь створити API, за допомогою якого інші програмісти-користувачі даного API зможуть створити нові клієнти для першого застосування, нові сервіси на його основі і, можливо, нададуть йому нового функціоналу.

- Розділення серверної (backend) та клієнтської (frontend) частин web-систем. Сутність розділення в тому, що для обох частин використовуються суттєво різні технології та мови. Серверна частина пишеться мовами C#, Java, Perl, PHP, тощо. Для клієнтської частини, що виконується в браузерах, є прийнятним лише HTML та javaScript і його фреймворки, наприклад, Angular чи React. Сумістити такі полярні технології можливо за умови існування у даних частин API та міжплатформених типів даних, таких як XML, JSON.

Отже, API компонентів та програмних систем має велике значення для їх сполучання і сумісного розвитку.

1.2. Застосування ручного та модульного тестування для верифікації інтерфейсів

Кожен окремий модуль являє собою окрему виділену частину коду програмного продукту, яка відповідає за роботу деякої складової загального функціоналу. Це є певні обчислення, операції бізнес-логіки (домену програмного забезпечення), обробка даних, взаємодія з джерелами даних програмного забезпечення, що розроблюється, або за його межами. Для тестування роботи таких програмних частин використовується як ручне тестування, що зазвичай виконується окремою людиною – тестувальником, так і тестування, що виконується спеціальними засобами (інструментами автоматизованого тестування, наприклад, сервер побудови готового коду – Build Server).

Ручне тестування – вид тестування, до якого залучається людина. Тестувальник імітує роботу кінцевого користувача, проходить основні бізнес-

сценарії згідно наданим інтерфейсам, які реалізовані у системі та офіційно задокументовані вручну. Такий вид тестування є ефективним у тестуванні будь-яких інтерфейсів, адже головна ціль – зробити продукт зрозумілим та корисним для кінцевого користувача. Але він має певні недоліки:

- Від збільшення кількості інтерфейсів та кількості сценаріїв до них збільшується і кількість роботи для тестувальника, а це означає великі часові витрати.
- Загострюється проблема людського фактору. Тестувальник не застрахований від помилки та непорозумінь.
- При нарощуванні функціоналу зростає складність тестування. Чим складніші вихідні дані та більший їх об'єм, тим важче людині (тестувальнику) з ними працювати.
- Якщо функціонал, що тестується, не має людино-машинного інтерфейсу (різного виду служби та web-сервіси), тестувальнику потрібно володіти інструментами побудови та виконання запитів до програмного забезпечення, що тестується (наприклад, підготовка даних для http-запиту, створення запиту, його відправлення, отримання результатів запиту та їх аналіз. Засобами підготовки та виконання таких запитів є Fiddler, Postman та ін.). Тобто для тестування програмних інтерфейсів необхідні певні професійні навички та знання.

Модульне тестування – вид тестування, який застосовується до вихідного коду, написаного програмістом. Таке тестування перевіряє, чи виконує свої функції кожна окрема мала частина програми (зазвичай, метод) належним чином. Ручне тестування в даному випадку задіяти неможливо, тому що ручний тестувальник не має доступу до вихідного коду та не має розуміння, яким чином код працює з середини, тому протестувати такі окремі елементи коду не може. Тому модульне тестування виконується кодом (модульними тестами), написаним програмістами додатково до основного коду. Зазвичай такі тести короткі і тестують окремі методи цільового коду, зрідка – їх взаємодію. Для імітації взаємодії створюються та використовуються методи-замінники оригінального функціоналу, що зменшує

залежність методів від зовнішніх даних. Отже метод, що тестується, може добре працювати з методом-замінником, але відбудеться помилка в реальних умовах. Тому не дивлячись на те, що модульні тести можна розробити для покриття всього функціоналу продукту, тести будуть запускатись на сервері побудови коду автоматично та наявність ручного тестувальника не обов'язкова, але такого виду тестування недостатньо для перевірки роботи певного функціоналу. Окрім вищеозначеного модульне тестування має додаткові недоліки:

- Недостатня якість тестування – модульні тести пишуться на код тією самою людиною, яка цей код написала, а отже не виключена ситуація, коли не всі ділянки коду покриваються модульним тестуванням.
- Обмеженість самого тесту у функціоналі – так як модульне тестування застосовується лише до окремих методів, воно не здатне перевірити правильність роботи бізнес функціоналу в загальному вигляді.

З огляду на ці недоліки модульне тестування не здатне перевірити взаємодію двох модулів, так як є локальним видом тестування.

Виконавши аналіз існуючих загально відомих та поширених видів тестування, можна зробити висновок, що потрібно розробити та запропонувати до використання нові види верифікації для тестування програмних інтерфейсів, щоб задовольнити наступні вимоги до тестування:

- Тестування має бути автоматизованим, тобто функції тестування має виконувати не людина, а програмний засіб. Таким чином не потрібно багато тестувальників, а лише комп'ютерна техніка. Це дозволить зменшити час на регресійне тестування і обробку великих масивів даних в декілька разів.
- Технології та бібліотеки, що при цьому будуть застосовуватися, не повинні бути занадто складними та важкими у засвоєнні принципів роботи з ними.

Отже, загальним висновком є те, що з однієї сторони потрібні нові засоби та види тестування, з іншої сторони потрібні інструменти, підтримка оволодіння ними та швидкого та простого застосування.

1.3. Актуальність розробки нових шляхів та засобів верифікації інтерфейсів слабо зв'язаних програмних компонентів

Системи, які складаються із слабо зв'язаних компонентів, мають значні переваги у розробці та їх розгортанні перед монолітними системами, а отже все частіше реалізація нових продуктів відбувається з залученням принципів функціонального програмування.

Новітні архітектури програмного забезпечення впроваджуються з урахуванням нових потреб користувачів. Все більше програмних продуктів мігрують до хмарних служб та сервісів, що дозволяє забезпечити безперебійність роботи та широку аудиторію користувачів. Архітектура з моноліту перетворюються на більш компонентну програмну систему. А це означає, що «спілкування», тобто взаємодію таких компонентів також потрібно передбачити. На сьогоднішній день, термін «інтерфейс» вийшов за рамки його трактування виключно в контексті графічного інтерфейсу продукту. Тоді природньо, що під «користувачем» також потрібно розуміти не тільки людину, що натискає клавіші, а й код, частину системи, що «використовує» інший код чи частину системи. У другому випадку інтерфейс – це API (програмний інтерфейс додатку), а користувачами є самі компоненти [14].

Наступне питання постає в тому, як впевнитися, що інтерфейс, який підтримує конкретний компонент, дійсно виконує свої функції та забезпечує безперебійний зв'язок з іншими компонентами. Таким чином, з'являється нова ланка в комплексі заходів з функціональної верифікації компонентів. Інтерфейс складається з точок доступу, які надсилають відповідь з певними даними на конкретний запит. Кожна така точка доступу має функціональні вимоги до того, які дані і у якому форматі вона «розуміє» та здатна обробити щоб виконати свою функціональність, повернувши створені нею нові дані певного змісту у певному форматі. Для опису такого співвідношення використовують контракт.

Контракт – це зафіксовані вимоги до структури і наповнення запитів та відповідей між конкретними двома точками доступу. Зазвичай для формування запитів та відповідей використовується існуючий веб-протокол.

Верифікація такого контракту має проходити у 2 етапи: з кожного компоненту окремо. Але постає проблема верифікації контракту, якщо компоненти, наприклад, знаходяться в стадії активної реалізації, чи щодо одного з компонентів застосуються структурні зміни, чи інший компонент вийшов з ладу та тимчасово є непрацюючим та недоступним. Відповіддю для рішення цієї проблеми є емуляція. Згідно запропонованого підходу кожен компонент має мати емулятор, який замість реального компоненту буде надсилати відповіді на визначені контрактом запити.

За типами запитів взаємодії компоненти можна поділити на дві групи: компоненти–користувачі та компоненти-постачальники. Компоненти-користувачі отримають дані, які надають компоненти-постачальники. Коли відбувається верифікація компонента-користувача, застосовується емулятор, який заміщує собою компонент-постачальник. І для перевірки, чи працює належним чином компонент-користувач, програмісту - його розробнику не потрібно очікувати наявності та працездатності компоненту, від якого компонент-користувач залежить, тобто компонента-постачальника. Таким чином завжди можливо перевірити, що даний компонент здатен приймати та обробляти дані формату, прописаного в контракті. Коли відбувається верифікація компонента-постачальника, застосовується емулятор, який заміщує собою компонент-користувач. Отримати відповідь, що постачальник працює коректно, стає можливим навіть у випадку, якщо ще не існує жодного клієнта на послуги, які виконує компонент-постачальник. Таким чином в будь-який час можливо перевірити, що даний компонент відправляє дані у заданому контрактом форматі.

Отже, можливо провести функціональну верифікацію інтерфейсу програмної системи без її повної готовності (і це є дуже суттєвим та важливим), або в моменти важливих структурних змін кожного з компонентів. Тому такий запропонований у дипломній роботі підхід до тестування, який підлягає

формалізації та розробки до нього супроводжуючих засобів, надає переваги у швидкості та виявлення дефектів на ранніх стадіях розробки.

1.4. Висновки по розділу 1

Для вирішення поставленої у магістерській роботі задачі верифікації інтерфейсів у системах, архітектура котрих побудована на принципі слабого зв’язування окремих компонентів системи, у першому розділі виконано наступні завдання:

- розкрито термін API програмних систем та їх окремих модулів і визначено роль API у написанні коду та його застосуванні;
- виявлено необхідність тестування API;
- розглянуто існуючі шляхи та методи тестування, які можливо застосувати у верифікації, їх відмінності та недоліки в застосуванні;
- доведено необхідність формування нового підходу до тестування API та розробки засобів підтримки виконання верифікації інтерфейсів слабо зв’язаних програмних компонентів за розробленим підходом.

2. Аналіз підходів та технологій до функціональної верифікації інтерфейсів програмної системи

2.1. Принципи взаємодії та налагодження компонентів у мікросервісній архітектурі

"Мікросервіс" - ще одна з сучасних парадигм у галузі архітектури програмного забезпечення. [3]

Архітектурний стиль мікросервісу - це підхід до розробки єдиної програми як сукупності малих сервісів, кожен з яких працює у власному процесі та спілкується з легкими механізмами, у якості яких досить часто виступає API ресурсів HTTP. Дані служби будуються на основі бізнес-можливостей і незалежно розгортаються за допомогою повністю автоматизованого механізму розгортання та налагодження. Існує мінімальне централізоване управління такими службами, які можуть бути написані на різних мовах програмування та використовувати різні технології зберігання даних.

Для глибокого та детального аналізу стилю мікросервісу (рис 2.1) корисно порівняти його з монолітним стилем: монолітний додаток вибудовується як єдине ціле. Прикладні програми підприємства часто складаються з трьох основних частин: користувальницький інтерфейс на стороні клієнта (який складається з HTML-сторінок та коду на JavaScript, що запускається в браузері на машині користувача); база даних (що складається з багатьох таблиць, розташованих в загальному сховищі під керуванням, як правило, реляційної системи управління базами даних), а також серверний додаток. Прикладна програма на сервері буде обробляти HTTP-запити, виконувати логіку домену, завантажувати та оновлювати дані з бази даних, а також вибирати і заповнювати HTML-представлення, що надсилаються браузеру. Такий серверний додаток є

монолітом - єдиний логічний виконуваний файл\модуль. Будь-які зміни в системі передбачають створення та розгортання нової версії додатка на сервері.

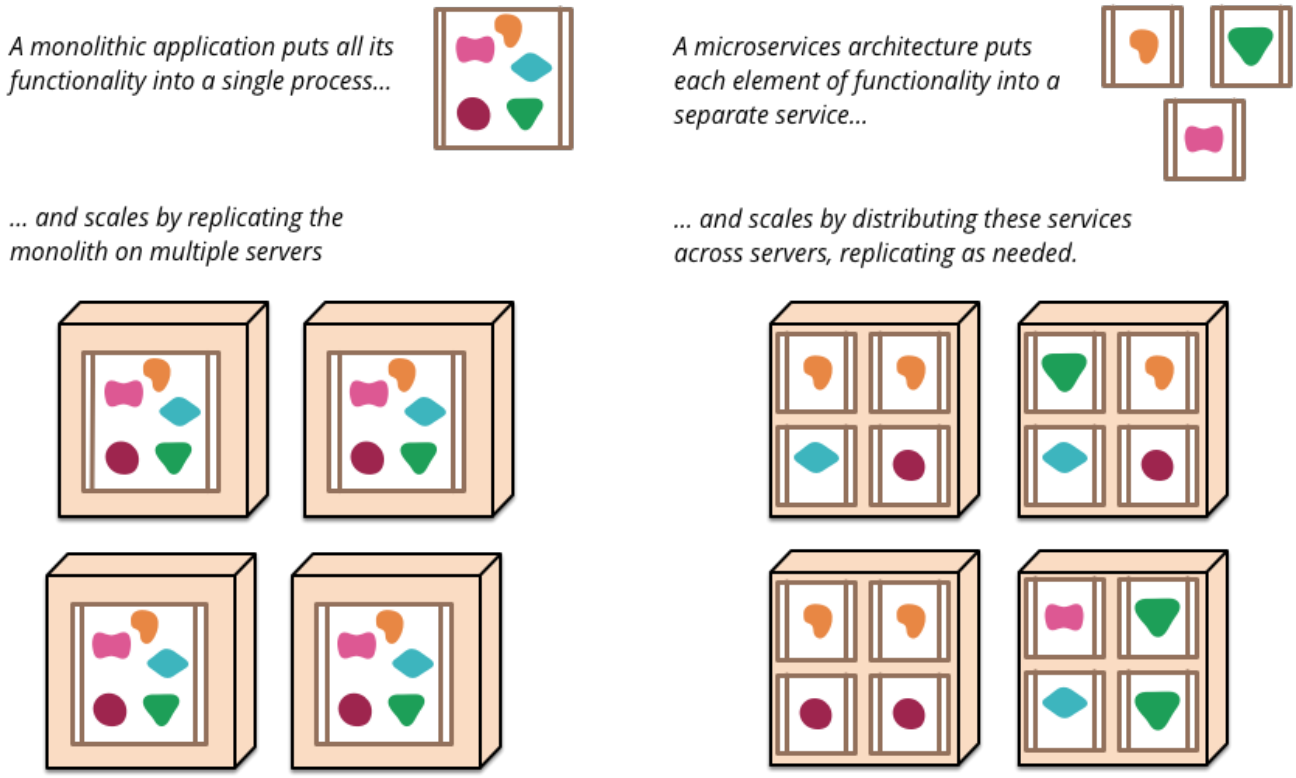

Рисунок 2.1 – Порівняння структури сервісів та мікросервісів

Монолітний сервер - це швидший та природний спосіб підійти до створення такої системи. Вся логіка обробки запиту виконується в одному процесі, що дозволяє використовувати основні функції мови для поділу програми на класи, функції та простори імен. Забезпечуючи певну обережність, можливо запустити та протестувати програму на техніці розробника, а також використовувати процедуру розгортання, щоб переконатися, що зміни правильно перевірені та використані у виробництві. Також є можливим горизонтально масштабувати моноліт, запускаючи багато потоків для балансування навантаження.

Монолітні програми можуть бути успішними, але по міру того, як все більше додатків розгортаються в хмарі, моноліти уступають більш сучасним архітектурним рішенням. До того ж з часом важко зберегти чітку та прозору модульну структуру, і це ускладнює збереження змін, які повинні впливати лише на один модуль у межах цього модуля. Масштабування вимагає масштабування всього додатку, а не його частини, що вимагає більшого ресурсу.

Ці недоліки монолітних додатків призвели до появи та поширення використання архітектурного стилю мікросервісу: створення додатків як провайдерів послуг. Окрім того, що послуги незалежно розгортаються та масштабуються, кожен сервіс також забезпечує кордон модуля, додатково дозволяючи створювати різні служби на різних мовах програмування. Їх також можуть розробляти різні команди.

Для надання формального визначення архітектурного стилю мікросервісу, потрібно розглянути та порівняти загальні характеристики архітектур. Як і в будь-якому визначенні, що описує загальні характеристики, не всі архітектури мікросервісу мають потрібні характеристики, але очікується, що більшість мікросервісних архітектур мають більшість характеристик з таких, що вимагаються або бажані.

Говорячи про компоненти, слід мати на увазі комплексне визначення того, що робить компонент. Визначення полягає в тому, що компонент - це частина програмного забезпечення, яку можна самостійно замінити та оновлювати.

Мікросервісні архітектури використовуватимуть бібліотеки, але їх основним способом компонування власного програмного забезпечення є поділ їх на сервіси. Бібліотеки визначаються як компоненти, пов'язані з програмою, і називаються функціями викликів у пам'яті, тоді як сервіси є компонентами поза процесом, які взаємодіють із таким механізмом, як запит веб-служби або віддалений виклик процедури.

Однією з основних причин використання служб як компонентів (а не бібліотек) є те, що сервіси розгортаються незалежно. Якщо існує програма, яка складається з декількох сервісів в одному процесі, модифікація будь-якого окремого компоненту вимагає перерозподілу всієї програми. Але якщо ця програма розгортається на кілька сервісів, можна очікувати, що для багатьох змін в одній службі буде потрібно лише розгорнути цю службу. Це не є абсолютним, деякі зміни змінять інтерфейси служб, що призведе до певної координації, однак мета гарної архітектури мікросервісу - мінімізувати їх за допомогою узгоджених між службами механізмами еволюції у контрактах на обслуговування.

Іншим наслідком використання служб як компонентів є більш явний компонентний інтерфейс. Більшість мов не мають хорошого механізму для визначення явного опублікованого інтерфейсу. Часто лише документація та дисципліна не дозволяють клієнтам зламати компоненти інкапсуляції, що призводить до надмірно жорсткого зв'язку між компонентами. Служби уникають цього, використовуючи явні механізми віддалених викликів.

Використання таких сервісів має свої недоліки. Віддалені виклики коштують дорожче, тому віддалені програмні інтерфейси мають бути менш гнучкими у використанні, що часто незручно використовувати. Якщо потрібно змінити розподіл обов'язків між компонентами, такі зміни поведінки складніші, коли перетинається межа процесу.

У першому наближенні можливо спостерігати, що сервіси співвідносяться з процесами у середовищі виконання, але це лише перше наближення. Сервіс може складатись з кількох процесів, які завжди будуть розроблятися та розгортатися разом, наприклад, процес подання заявки та база даних, яка використовується лише цією службою.

Коли великий додаток потрібно розділити на частини, управлінці часто фокусуються на стеку технологій, в результаті чого з'являються команди з розробки користувацького інтерфейсу, команди з розробки бізнес-логіки на стороні сервера та команди з підтримки бази даних. Коли команди поділяються за цими принципами, навіть прості зміни можуть призвести до міжгрупового проекту, який вимагає схвалення часу та бюджету. І тоді кожна з команд зробить свою частину роботи лише на тому рівні, до якого мають доступ члени команди. Це приклад закону Конвей [12] у дії (рис 2.2).

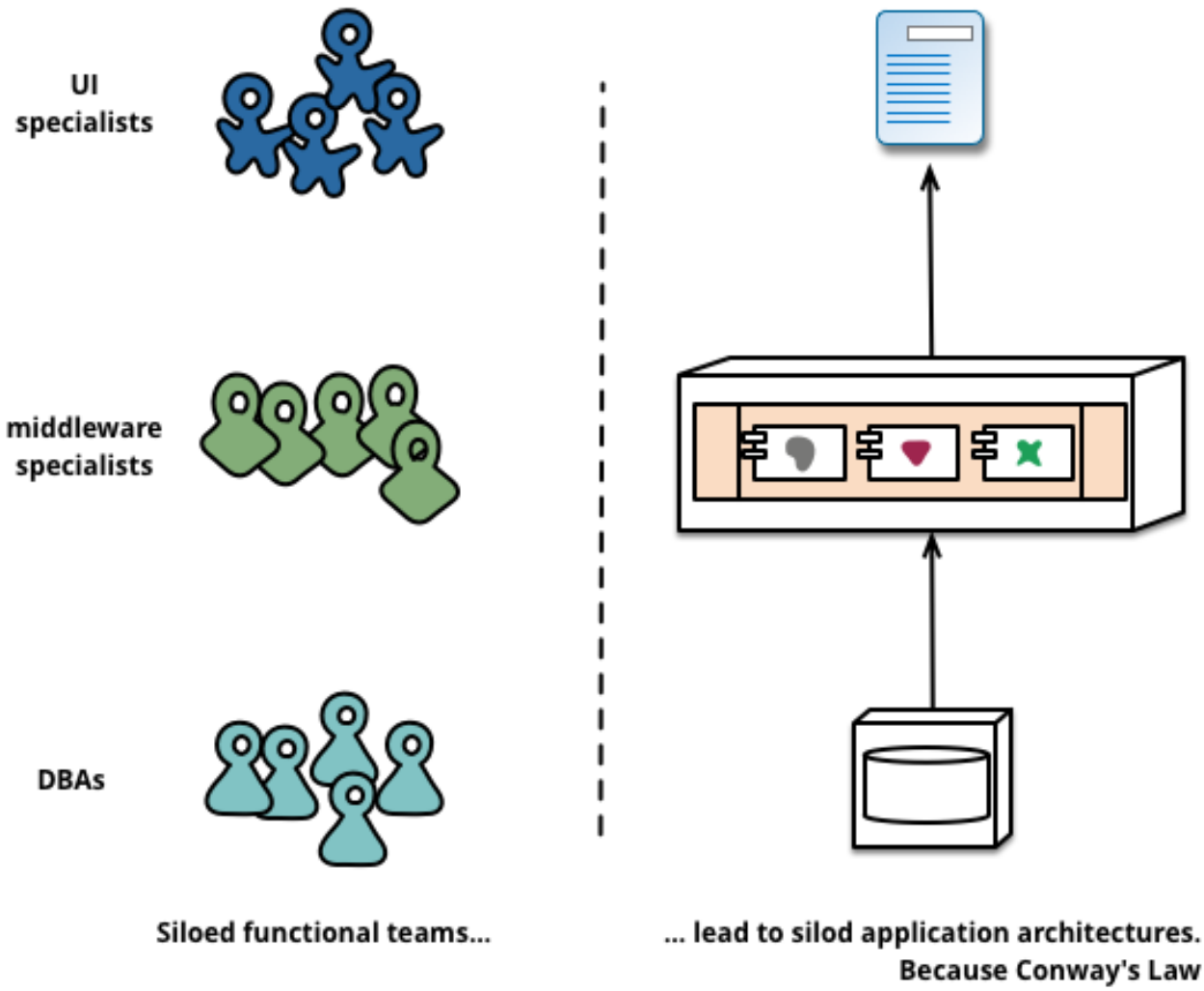

Рисунок 2.2 – Приклад закону Конвей

Мікросервісний підхід до поділу різний, поділяючись на сервіси, організовані навколо бізнес-можливостей. Такі сервіси використовують широкосмугове програмне забезпечення для цієї галузі бізнесу, включаючи користувальницький інтерфейс, постійне зберігання та будь-які зовнішні взаємодії. Таким чином, команди стають міжфункціональними, включаючи весь набір навичок, необхідних для розвитку: досвід користувачів, з бази даних та з управління проектами.

Міжфункціональні команди відповідають за створення та експлуатацію кожного продукту, і кожен продукт розділений на кілька окремих служб, які спілкуються через шину повідомлень.

Великі монолітні проекти завжди можуть бути модульними стосовно бізнес-можливостей, хоча це часто не так. Основна проблема полягає в тому, що як

правило, організовується занадто багато контекстів. Якщо моноліт охоплює багато з цих модульних кордонів, для окремих членів команди це може виявитись складним. Крім того модульні лінії вимагають великої дисципліни для забезпечення відповідності. Потрібне чітке розділення, необхідне для компонентів служби, щоб полегшити збереження меж кожної з команд.

Велика частина роботи над розробкою додатків використовує проектну модель: метою є надання певного програмного забезпечення, яке потім вважається завершеним. Після завершення, програмне забезпечення передається службовій організації, і команда проекту, яка її створила, розформується.

Прихильники мікросервісу, як правило, уникають цієї моделі, віддаючи перевагу думці, що команда повинна володіти продуктом протягом всього свого терміну служби. Команда розробників несе повну відповідальність за виробництво програмного забезпечення. Це призводить розробників до щоденного спілкування з тим, як їх програмне забезпечення веде себе у виробництві, і збільшує контакт з користувачами, оскільки вони повинні брати принаймні частину тягаря з підтримки.

Менталітет продукту пов'язаний з діловими можливостями. Замість того, щоб побачити програмне забезпечення як набір функціональних можливостей, які необхідно реалізувати, існує постійний взаємозв'язок, в якому йдеться про те, як програмне забезпечення може допомогти своїм користувачам розширювати свої бізнес-можливості.

Немає підстав, чому такий самий підхід не можна використовувати з монолітними додатками, але нижчий рівень деталізації послуг може спростити створення взаємин між розробниками послуг та їх користувачами.

Створюючи зв'язкові структури між різними процесами, було розглянуто багато продуктів та підходів, які підкреслюють необхідність інвестування значних ресурсів у сам механізм комунікації. Хорошим прикладом цього є Enterprise Service Bus (ESB), де ESB продукти часто містять складні інструменти для маршрутизації повідомлень, трансформації та застосування бізнес-правил.

Але кращим виявився інший підхід: інтелектуальні кінцеві точки та прості транспортні канали. Програми, створені на основі принципів мікросервісності, підключаються і взаємопов'язані, вони володіють власною логікою домену та діють більше як фільтри у сенсі класичного Unix - вони отримують запит, застосовують відповідну логіку та створюють відповідь. Такі програми створюються за допомогою простих протоколів REST, а не складних протоколів, таких як BPEL.

Найчастіше використовуються два протоколи: HTTP-запит-відповідь з API ресурсів і легкий обмін повідомленнями.

Команди використовують принципи та протоколи, на яких побудована всесвітня мережа. Часто використовувані ресурси, які легко можна кешувати розробниками або спеціалістами з технічного обслуговування.

Другий підхід у загальному використанні - це обмін повідомленнями через спрощену шину повідомлень. Вибрана інфраструктура, як правило, є простою (наприклад, працює тільки як маршрутизатор повідомлень) - прості реалізації, такі як RabbitMQ або ZeroMQ, не роблять набагато більше, ніж забезпечують надійну асинхронну структуру.

У моноліті компоненти виконуються у процесі, і зв'язок між ними здійснюється шляхом виклику методу або виклику функції. Найбільшою проблемою перетворення моноліту в мікросервіс є зміна схеми зв'язку. Перетворення з виклику методу in-memory до RPC призводить до громіздкої комунікації, що є неефективним.

Одним із наслідків централізованого управління є тенденція до стандартизації на одній технологічні платформі.

Розділяючи компоненти моноліту на сервіси, з’являється вибір технологій і мов при створенні кожного з них.

Команди, які будують мікрослужби, також віддають перевагу іншим підходам до стандартів. Замість того, щоб використовувати набір конкретних стандартів, вони віддають перевагу ідеї створення корисних інструментів, які інші розробники можуть використовувати для вирішення проблем, аналогічних тим, з

якими вони стикаються. Ці інструменти, як правило, збираються з реалізації та використовуються разом з більш широкою групою, іноді, але не тільки з використанням внутрішньої моделі з відкритим вихідним кодом. Тепер, коли Git і GitHub стали де-факто системою керування версіями, практика відкритого коду стає все більш поширеною в рамках компанії.

Netflix є гарним прикладом організації, яка слідує за цією філософією. Спільні бібліотеки, як правило, орієнтовані на спільні проблеми зберігання даних та автоматизацію інфраструктури.

Для мікросервісного співтовариства накладні витрати особливо неприваблівi. Це не означає, що громада не оцінює сервісні контракти. Навпаки, вони, як правило, набагато більші. Вони просто дивляться на різні способи управління цими контрактами. Такі шаблони, як Tolerant Reader [15] та Consumer-Driven Contracts [16], часто застосовуються до мікросервісу. Виконання контрактів на замовлення в рамках вашої збірки підвищує довіру та забезпечує швидкий зворотний зв'язок щодо функціонування ваших сервісів. Це стає частиною автоматичної побудови ще до того, як буде написано код для нової служби. Тоді сервіс створюється лише настільки, що він задовольняє умовам контракту - підхід, який дозволяє уникати дилеми "YAGNI" [17] при створенні нового програмного забезпечення. Ці методи та інструменти, що утворюються навколо них, обмежують необхідність централізованого управління контрактами, зменшуючи тимчасовий зв'язок між службами.

Можливо, апогей децентралізованого управління є принципом його побудови/запуску. Команди несуть відповідальність за всі аспекти програмного забезпечення, яке вони створюють, включаючи роботу програмного забезпечення 24 години на добу. Перенесення цього рівня відповідальності, безумовно, не є нормою, але спостерігається, що все більше і більше компаній покладають відповідальність на команди розробників. Netflix є іншою організацією, яка прийняла цей принцип. Ці ідеї далеко від традиційної централізованої моделі управління, наскільки це можливо.

Децентралізація управління даними представлена різними способами (рис 2.3). На самому абстрактному рівні це означає, що концептуальна модель світу відрізнятиметься між системами. Це загальна проблема з інтеграцією у великому підприємстві, презентація продажів клієнтів буде відрізнятися від підтримки кінцевих споживачів.

Ця проблема є спільною для додатків, але може також виникати в програмах, особливо якщо ця програма розділена на окремі компоненти - це є основою поняття "Обмежений контекст", керований доменом. DDD розбиває складний домен на кілька обмежених контекстів і відображає зв'язки між ними. Цей процес корисний як для монолітних, так і для мікросервісних архітектур, але існує природна кореляція між межами сервісу та контекстом, що допомагає уточнити, і це зміцнює поділ.

На додаток до децентралізації рішень на концептуальних моделях, мікросервіси також децентралізують рішення щодо зберігання даних. Хоча монолітні програми віддають перевагу одній логічній базі даних для постійних даних, корпоративні системи часто віддають перевагу одній базі даних для ряду додатків - багато з цих рішень базуються на комерційних моделях вендорів.

Децентралізація відповідальності за дані за допомогою мікросервісу є важливою для управління оновленнями. Загальний підхід до роботи з оновленнями - це використання транзакцій для забезпечення узгодженості при оновленні декількох ресурсів. Цей підхід часто використовується в монолітах.

Використання таких транзакцій допомагає забезпечити послідовність, але накладає суттєвий тимчасовий зв'язок, що є проблематичним для кількох служб. Розподілені транзакції, як правило, важко реалізувати, і, як наслідок, архітектура мікросервісів підкреслює не-транзакційну координацію між службами, з чітким визнанням того, що узгодженість може бути лише можливістю узгодженості, а проблеми можуть бути вирішені компенсаційними операціями.

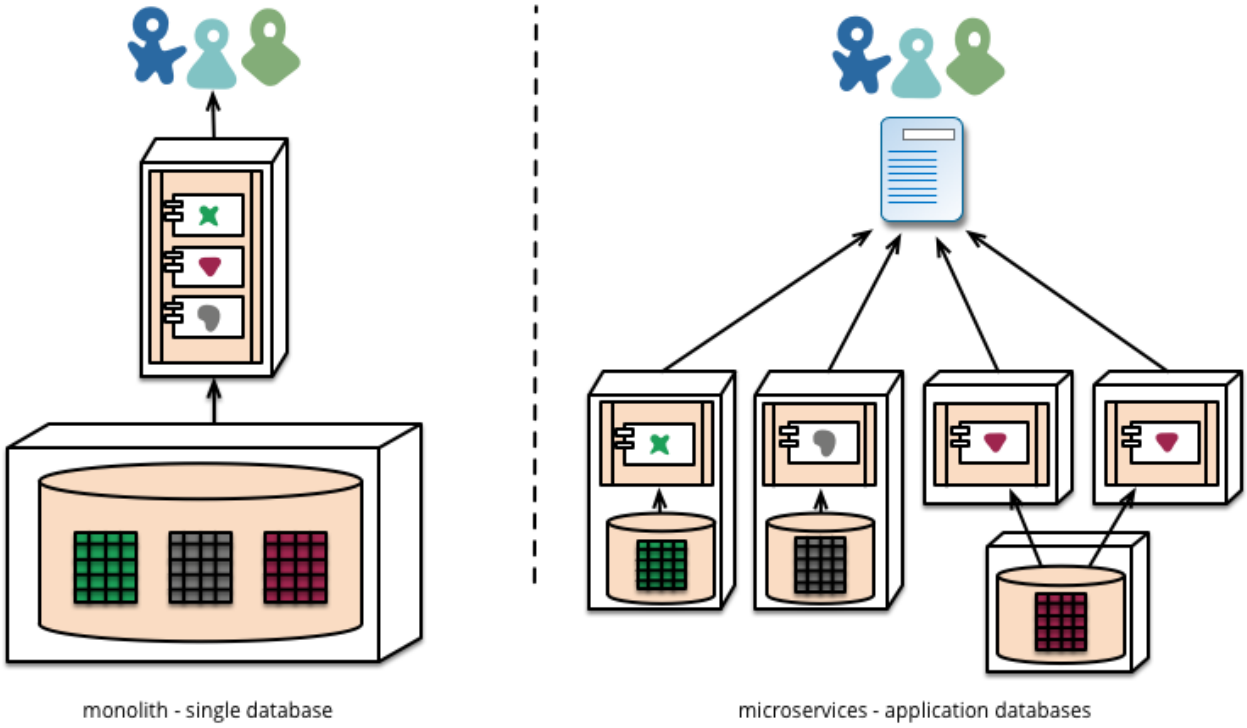

Рисунок 2.3 – Децентралізація даних у мікросервісах

Таким чином, вибір керувати невідповідністю є новим викликом для багатьох груп розвитку, але часто це відповідає діловій практиці. Часто компанії стикаються з певним ступенем непослідовності, щоб швидко реагувати на попит, в той же час вони мають певний зворотній процес для усунення помилок. Такий компроміс того вартує, якщо вартість виправлення помилок менша за вартість втраченого бізнесу з більшою послідовністю.

Наслідком використання служб як компонентів є те, що додатки повинні бути розроблені таким чином, щоб вони могли витримувати сервісні відмови. Будь-який запит до сервісу може бути невдалим через відсутність постачальника, клієнт повинен відповісти на нього як можна витончено. Це недолік у порівнянні з монолітним дизайном, оскільки він створює додаткову складність у його обробці. Наслідком цього є те, що команди розробки мікросервісу постійно мають приймати до уваги те, як сервісні помилки впливають на роботу користувачів.

Оскільки сервіси можуть вийти з ладу в будь-який час, важливо швидко мати змогу виявити збої та, якщо це можливо, автоматично відновити службу.

Команди розробника багатокомпонентної системи приділяють велику увагу моніторингу програми в режимі реального часу, перевіряючи як архітектурні

елементи (скільки запитів база даних отримує в секунду) так і відповідні бізнес-показники (наприклад, скільки замовлень приймається за хвилину). Семантичний моніторинг може забезпечити систему раннього попередження про те, що щось не так, що спонукає команди розробників стежити та досліджувати.

Це особливо важливо для архітектури мікросервісу, оскільки перевага спільна робота подій [18] призводить до появи поведінки. Моніторинг є життєво важливим для швидкого виявлення несприятливого поведінки, що може призвести до виправлення ситуації.

Моноліт можна побудувати так само прозоро, як мікросервіс. Різниця в тому, що абсолютно точно необхідно знати, коли служби, що працюють в різних процесах, відключені.

Команди очікують побачити складні параметри моніторингу та реєстрації для кожного окремого сервісу, наприклад інформаційні панелі, що показують статус "вгору/вниз", а також різні операційні та бізнес-показники. Детальні відомості про стан, поточної пропускної здатності та затримки є іншими прикладами, з якими ми часто стикаємося в реальному світі.

Фахівці мікросервісів, як правило, мають досвід еволюційного дизайну та розглядають декомпозицію служб як інший інструмент, який дозволяє розробникам програм контролювати зміни в їх застосуванні без уповільнення змін. Керування змінами не обов'язково означає зменшення змін - за допомогою правильних налаштувань та інструментів часто ви можете швидко і добре стежити за змінами програмного забезпечення.

Кожного разу, коли приймається рішення поділити програмну систему на компоненти, постає питання у тому як розділити фрагменти, на яких принципах. Ключовою особливістю компоненту є концепція незалежної заміни та можливість оновлення, що означає, що шукаються фрагменти, які можливо переписати, не впливаючи на його співавторів.

Веб-сайт Guardian є гарним прикладом програми, яка була розроблена та побудована як моноліт, але розвивається до мікросервісу. Моноліт як і раніше є основою сайту, але вони вважають за краще додавати нові функції, створюючи

мікросервіси за допомогою API monolith. Цей підхід особливо корисний для тимчасових функцій, таких як спеціалізовані сторінки для обробки спортивних подій. Цю частину веб-сайту можна швидко зібрати, використовуючи мови швидкого розвитку та видаляти після завершення події.

Такий акцент на взаємозамінності є особливим випадком більш загальним принципом модульної конструкції, яка полягає у керуванні модульністю за допомогою шаблону змін. Частини системи, які рідко змінюються, повинні бути в різних службах, ніж ті, які в даний час зазнають значного відтоку.

Впровадження компонентів у службі додає можливості більш детального планування випуску. З монолітом будь-які зміни вимагають нового повного розгортання всієї програми. Однак у випадку з мікрослужбовцями потрібно лише перевпорядковувати ті служби, які ви змінили. Це може спростити та прискорити процес випуску. Недоліком є те, що вам потрібно турбуватися про зміни в одній службі, яка може порушити її споживачів. Традиційний інтеграційний підхід полягає в тому, щоб спробувати вирішити цю проблему, використовуючи контроль версій, але в світі мікросервісів перевага полягає в тому, щоб використовувати контроль версій лише як крайній засіб. Ми можемо уникнути великої кількості версій, розвиваючи послуги, щоб вони були максимально стійкими до змін у своїх постачальників.

Архітектурний стиль мікросервісу є важливою ідеєю, яка заслуговує серйозного розгляду для корпоративних програм. Мікрослужби - це майбутній напрям для програмних архітектур.

Є, звичайно, причини, чому можна очікувати, що мікросервіс погано розвинеться. У будь-якому прагненні компонувати, успіх залежить від того, наскільки добре програмне забезпечення входить до компонентів. Тут важко точно визначити, де повинні лежати межі компонента. Еволюційний дизайн визнає труднощі належного визначення меж та важливості їх легко реорганізувати. Але коли компонентами є сервіси з віддаленим зв'язком, рефакторинг набагато складніший, ніж у бібліотеках. Переміщення коду ускладнене, а будь-які зміни в інтерфейсі повинні бути послідовними між учасниками.

2.2. Побудова інтерфейсів мікросервісів за принципом REST

REST (Representational state transfer) - це стиль архітектури програмного забезпечення для розподілених систем, таких як World Wide Web, який, як правило, використовується для побудови веб-служб. Системи, що підтримують REST, називаються RESTful-системами [6].

У загальному випадку REST є дуже простим інтерфейсом управління інформацією без використання додаткових внутрішніх прошарків. Кожна одиниця інформації однозначно визначається глобальним ідентифікатором, таким як URL. Кожна URL в свою чергу має строго заданий формат.

Відсутність додаткових внутрішніх прошарків означає передачу даних в тому ж вигляді, що і самі дані. Тобто дані не розгортаються в XML, як у SOAP і XML-RPC, не використовується AMF, як у Flash і т.д.

Кожна одиниця інформації однозначно визначається URL - це значить, що URL по суті є первинним ключем для одиниці даних. Тобто наприклад третя книга з книжкової полиці матиме вигляд /book/3, а 35 сторінка в цій книзі - /book/3/page/35. Звідси і виходить строго заданий формат. Причому абсолютно не має значення, в якому форматі знаходяться дані за адресою /book/3/page/35 - це може бути і HTML, і відсканована копія у вигляді jpeg-файлу, і документ Microsoft Word.

Управління інформацією сервісу цілком і повністю ґрунтується на протоколі передачі даних. Найбільш поширений протокол звичайно ж HTTP. Так ось, для HTTP дію над даними задається за допомогою методів: GET (отримати), PUT (додати, замінити), POST (додати, змінити, видалити), DELETE (видалити). Таким чином, дії CRUD (Create-Read-Updtae-Delete) можуть виконуватися як з усіма 4-ма методами, так і тільки за допомогою GET і POST. Взагалі, POST може використовуватися одночасно для всіх дій зміни. Це дозволяє іноді обходити неприємні моменти, пов'язані з неприйняттям PUT і DELETE.

Як відомо, web-сервіс - це додаток працює в World Wide Web і доступ до якого надається по HTTP-протоколу, а обмін інформацією йде за допомогою формату XML. Отже, формат даних переданих в тілі запиту буде завжди XML.

Для кожної одиниці інформації (info) визначається 5 дій. А саме:

GET/info/(Index) - отримує список всіх об'єктів. Як правило, це спрощений список, тобто містить тільки поля ідентифікатора і назви об'єкта, без інших даних.

GET/info/{id} (View) - отримує повну інформацію про об'єкті.

PUT/info/або POST/info/(Create) - створює новий об'єкт. Дані передаються в тілі запиту без застосування кодування, навіть urlencode. У PHP тіло запиту може бути отримано таким способом:

POST/info/{id} або PUT/info/{id} (Edit) - змінює дані з ідентифікатором {id}, можливо замінює їх. Дані так само передаються в тілі запиту, але на відміну від PUT тут є певний нюанс. Справа в тому, що POST-запит має на увазі наявність urldecoded-post-data. Тобто якщо не застосовувати кодування - це порушення стандарту. Тут хто як хоче - деякі не звертають уваги на стандарт, деякі використовують якусь post-змінну.

DELETE/info/{id} (Delete) - видаляє дані з ідентифікатором {id}.

Ще раз зазначу, що в нашому прикладі/info/- може базуватися на якійсь іншій інформації, що може бути (і повинно) бути відображено в URL.

Як видно, в архітектура REST дуже проста в плані використання. По виду запиту відразу можна визначити, що він робить, не розбираючись в форматах (на відміну від SOAP, XML-RPC). Дані передаються без застосування додаткових шарів, тому REST вважається менш ресурсномістких, оскільки не треба парсити запит щоб зрозуміти що він повинен зробити і не треба переводити дані з одного формату в інший.

2.3. Верифікація інтерфейсу мікросервісної архітектури

Тестування контрактів - це спосіб переконатися, що компоненти (наприклад, постачальник програмного інтерфейсу та клієнт цього інтерфейсу) можуть взаємодіяти один з одним очікуваним шляхом. Без такої перевірки контракту єдиним способом визначити, що компоненти можуть взаємодіяти, є використання дорогих та крихких інтеграційних тестів. Отже, контрактне тестування, яке пропонується у дипломній роботі до використання, дозволяє підтвердити, що ваші компоненти працюватимуть разом без необхідності повноцінного розгортання.

Для того, щоб ефективно застосувати тестування контрактів, потрібно належним чином проаналізувати його особливості та схему застосування. Тому далі надається відповідний опис.

2.3.1. Умови застосування контрактного тестування

Тестування контрактів призначено для ситуації, коли існують два компоненти, які потребують взаємодії, наприклад API клієнта та веб-інтерфейс. І контрактне тестування також працює в середовищі з багатьма сервісами (широко розповсюджений випадок архітектури мікросервісів) [11].

Як правило, контракт створюється між споживачем (наприклад, клієнтом, який хоче отримати деякі дані) (рис 2.4) і постачальником (наприклад, API на сервері, який надає дані, необхідні для клієнта) (рис 2.5). У мікросервісних архітектурах традиційні терміни «клієнт» і «сервер» не завжди доречні - наприклад, коли зв'язок досягається через черги повідомлень. З цієї причини введено нові терміни-аналоги «споживач» та «постачальник».

Тестування контрактів відбувається «на замовлення». Це означає, що написаний контракт є частиною тестування споживача. Основною перевагою такого підходу є те, що перевіряються лише ті частини комунікації, які фактично використовуються споживачем. Це, у свою чергу, означає, що будь-яка поведінка

постачальника, яка не використовується нинішніми споживачами, може змінюватися без порушення контрактів.

На відміну від схеми або специфікації, які є статичними артефактами, що описують всі можливі стани ресурсів, контракт реалізується шляхом виконання набору тестових випадків, кожен з яких описує одну конкретну пару запитів/відповідей.

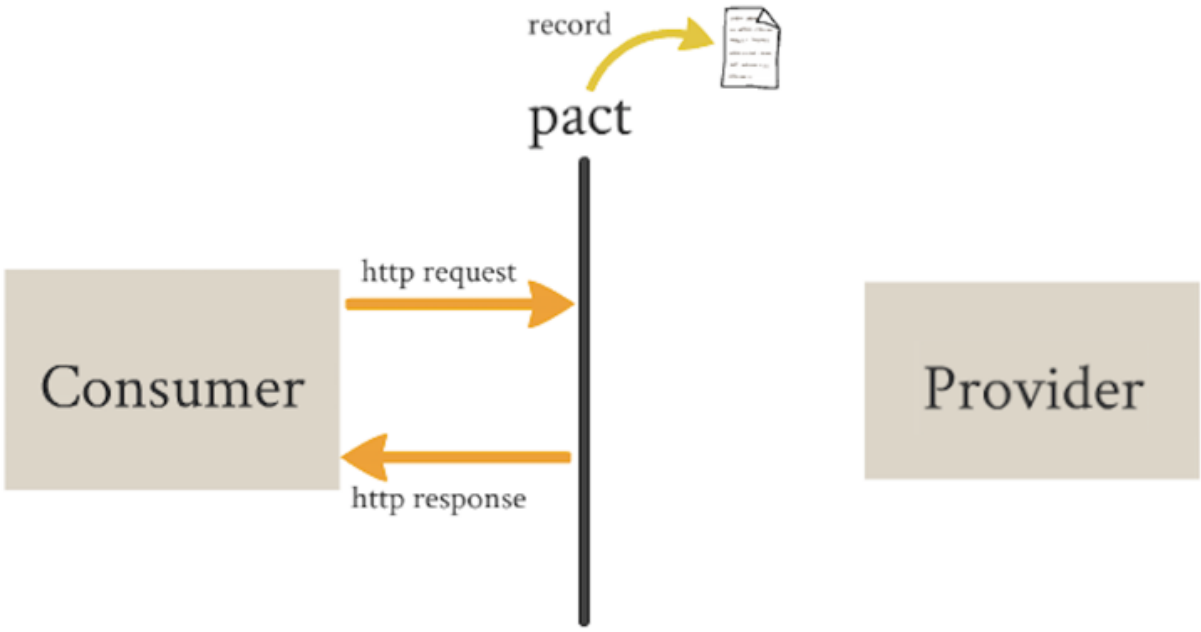

Рисунок 2.4 – процедура контрактного тестування для користувача

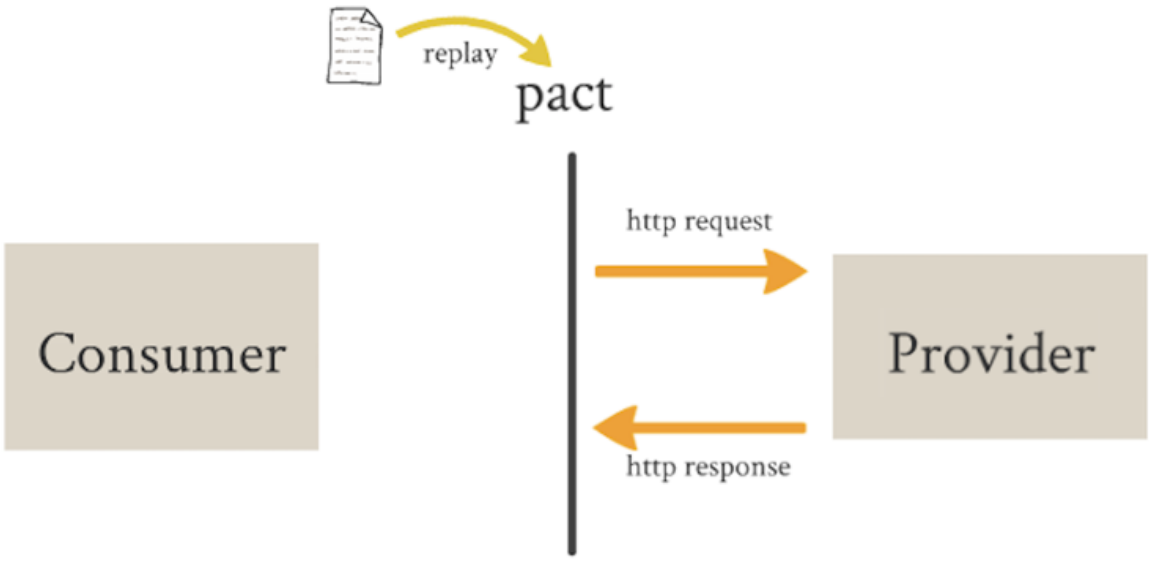

Рисунок 2.5 – процедура контрактного тестування для користувача

2.3.2 Тестування постачальника та споживача

Різниця між контрактними тестами та функціональними тестами є інколи досить невизначеною, так як це не є тестуванням чорної скриньки чи тестуванням вихідного коду безпосередньо, крім того контрактне тестування залежить від глибини зв'язків, що описують контракти.

Одна з таких залежностей – це перевірки та відхилені запити. Наприклад, є деякий простий сервіс, який дозволяє споживачам зареєструвати нових користувачів, як правило, за допомогою запиту POST, що містить інформацію про створеного користувача в тілі запиту.

Простий сценарій "успішного шляху" для цієї взаємодії може виглядати наступним чином:

```
Given "there is no user called Mary"
When "creating a user with username Mary"
  POST /users { "username": "mary", email: "...", ... }
Then
  Expected Response is 200 OK
```

Дотримуючись "успішних шляхів", є ризик втратити різні коди відповідей, можливо, через нерозуміння споживачами поведінки постачальника. Тож розглянемо сценарій відмови:

```
Given "there is already a user called Mary"
When "creating a user with username Mary"
  POST /users { "username": "mary", email: "...", ... }
Then
  Expected Response is 409 Conflict
```

Контракт має не залежати від сценарію і бути максимально універсальним. Це забезпечить стабільність тестів, адже враховується лише структура запитів/відповідей. Наприклад:

```
When "creating a user with an invalid username"
  POST /users { "username": "bad_username_that_breaks_some_rule_that_you_are_fairly_confident_will_not_change", ... }
Then
  Response is 400 Bad Request
```

```
Response body is { "error": "<any string>" }
```

Контракти повинні стосуватися лише:

- дефектів у споживачі;
- неправильного розуміння споживачами точок доступу;
- змін у постачальника і його точок доступу

Контракти не повинні бути залежними від бізнес-логіки, але повинні підтверджувати, що споживач та постачальник мають спільне розуміння того, які запити та відповіді надсилаються та очікуються. Отже сутність контрактів в тому, що перевіряється саме спосіб взаємодії, а не ціль та результат взаємодії (виконання бізнес-операцій, отримання даних, тощо).

Мета контрактного тестування полягає у забезпеченні того, щоб споживач і постачальник мали спільне розуміння повідомлень, які будуть передані між ними. Коли перевіряється контракт, це не тільки гарантія, що постачальник поверне очікувані дані споживачеві, а також перевірка, чи споживач правильно використовує програмний інтерфейс.

Перевірка того, що програмний інтерфейс використовується правильно в контрактних тестах, може бути дещо важким, оскільки постачальник часто ігнорує неправильний ввід даних і не видає помилки.

Класичним прикладом цього є те, що споживач користується правильними параметрами запиту для пошуку. Параметри запиту з недійсними іменами, ймовірно, будуть проігноровані, що може призвести до некорректних відповідей у кінцевих версіях в майбутньому.

Найпростіше рішення для забезпечення правильності параметрів, які використовуються для запиту, полягає в тому, щоб постачальник відображав параметри, які він використовував у запиту, у своїй відповіді.

Це рішення передбачає співпрацю обох сторін договору. Це не означає пропозицію використати такі контракти для функціонального тестування. Ці тести належать до вихідного коду постачальника. Але в такому випадку, на жаль, єдиним способом перевірки використання правильного імені параметра є перевірка повернутих результатів.

2.4. Обгрунтування контрактного тестування як підходу до функціональної верифікації інтерфейсів програмних систем

Тестування програмного забезпечення - перевірка відповідності між реальним і очікуваним поведінкою програми, що здійснюється на кінцевому наборі тестів, обраному певним чином. У більш широкому сенсі, тестування - це одна з технік контролю якості, що включає в себе активності з планування робіт (Test Management), проектування тестів (Test Design), виконання тестування (Test Execution) і аналізу отриманих результатів (Test Analysis).

Якість програмного забезпечення (Software Quality) - це сукупність характеристик програмного забезпечення, що відносяться до його здатності задовольняти встановлені і передбачувані потреби [5].

Верифікація (verification) - це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умов, сформованим на початку цього етапу. Тобто чи виконуються наші цілі, терміни, завдання по розробці проекту, визначені на початку поточної фази.

Цілі тестування:

- Підвищити ймовірність того, що додаток, призначений для тестування, буде працювати правильно при будь-яких обставинах.
- Підвищити ймовірність того, що додаток, призначений для тестування, буде відповідати всім описаним вимогам.
- Надання актуальної інформації про стан продукту на даний момент.

Дефект (bug) - це невідповідність фактичного результату виконання програми очікуваного результату. Дефекти виявляються на етапі тестування програмного забезпечення (ПО), коли тестувальник проводить порівняння отриманих результатів роботи програми (компонента або дизайну) з очікуваним результатом, описаним в специфікації вимог.

Функціональні види тестування:

- Функціональне тестування (Functional testing)
- Тестування інтерфейсу для користувача (GUI Testing)
- Тестування програмного інтерфейсу (API Testing)
- Тестування безпеки (Security and Access Control Testing)
- Тестування взаємодії (Interoperability Testing)

Необхідно визначити, яке місце і значення належить займати контрактному тестуванню у загальній класифікації видів тестувань.

В першому розділі дипломної роботи було розглянуто два шляхи, якими виконується сучасне тестування: мануальне (вручну) та автоматизоване тестування. З огляду на те, що при тестуванні програмних компонентів та їх взаємодії, якщо це стосується мікросервісів, користувацький інтерфейс відсутній, тому фокус уваги переноситься на автоматизоване тестування шляхом написання та виконання спеціального коду (тестів користувацького інтерфейсу, модульних тестів, інтеграційних тестів).

Тестову автоматизацію прийнято розглядати у вигляді піраміди. «Піраміда» (рис 2.6) – це найпоширеніша схема, яка використовується в технічній документації та спеціалізованій літературі [2]. Вона вважається стандартом, що пропонується як правильний та ефективний спосіб побудови та структурування автоматизованих тестів програмного забезпечення. Форма піраміди вказує не тільки на послідовність застосування тестів різних груп, але й їх відносну кількість. Так, відповідно до піраміди кількість модульних тестів повинна бути максимально можливою. Кількість інтеграційних (службових) тестів має бути дещо меншою. Що стосується тестів інтерфейсу користувача, їх кількість завжди повинна бути настільки мала, наскільки це можливо.

Для монолітних веб-додатків побудова та застосування автоматизованих тестів за принципом піраміда задовольняє всі потреби щодо автоматизованого тестування. Для мікрослужб застосування схеми піраміди має мати інший вигляд. Для того, щоб пристосувати «піраміду» до тестування мікросервісів, потрібно

визначити особливості їх функціонування, які впливатимуть на можливість та спосіб їх протестувати.

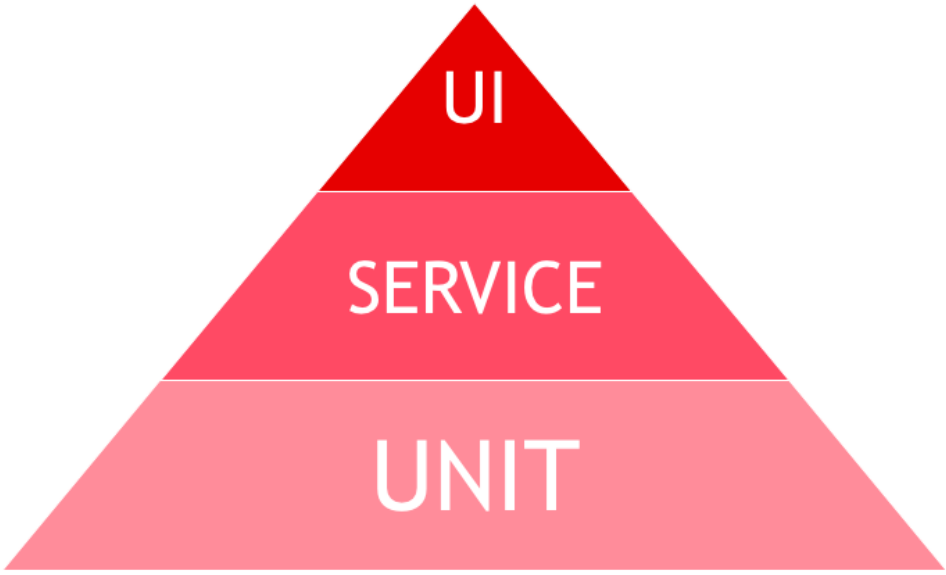

Рисунок 2.6– Схема тестових рівнів «піраміда»

Мікрослужби – це сучасний тип архітектури програмного забезпечення, що часто використовується для створення складних, слабо зв'язаних та надійних систем з потужною та розвинутою «бекенд»-частиною (рис 2.7).

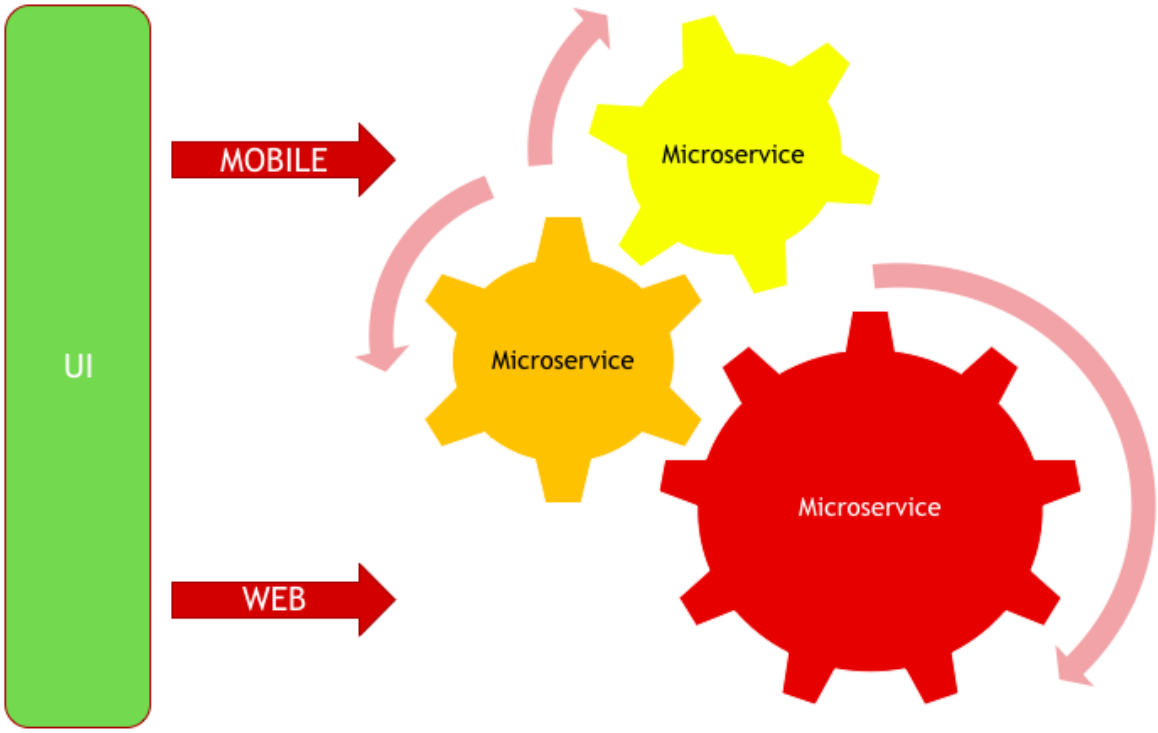

Рисунок 2.7 – Приклад взаємодії компонентів слабо зв'язаної системи

Тобто замість великої монолітної серверної системи, яка містить всю бізнес-логіку та код взаємодії з базою даних, використовується набір незалежних служб-компонентів, що розгортаються окремо один від одного, та надають у використання свою частину бізнес-логіки. Оскільки поставлено завдання автоматизувати тестування такі системи, потрібно переглянути поточний класичний підхід до структурування тестових рівнів.

Одже, тепер кожен мікросервіс має бути представлений як незалежна система, яка вимагає тестування на різних рівнях: модуль, інтеграція, компонент. Інтеграція між індивідуальними мікрослужбами повинна також охоплюватися інтеграційними тестами.

До того ж щоб впевнитись, що система в цілому також працює коректно, потрібно реалізувати кінцеві тести для рівнів API або інтерфейсу користувача.

Як наслідок, схема тестування «піраміда» зазнає трансформації при застосуванні її до тестування мікросервісу. Згідно з цією трансформованою схемою (рис 2.8), кожна мікрослужба перевіряється окремо (з урахуванням фактичних або фіктивних залежностей).

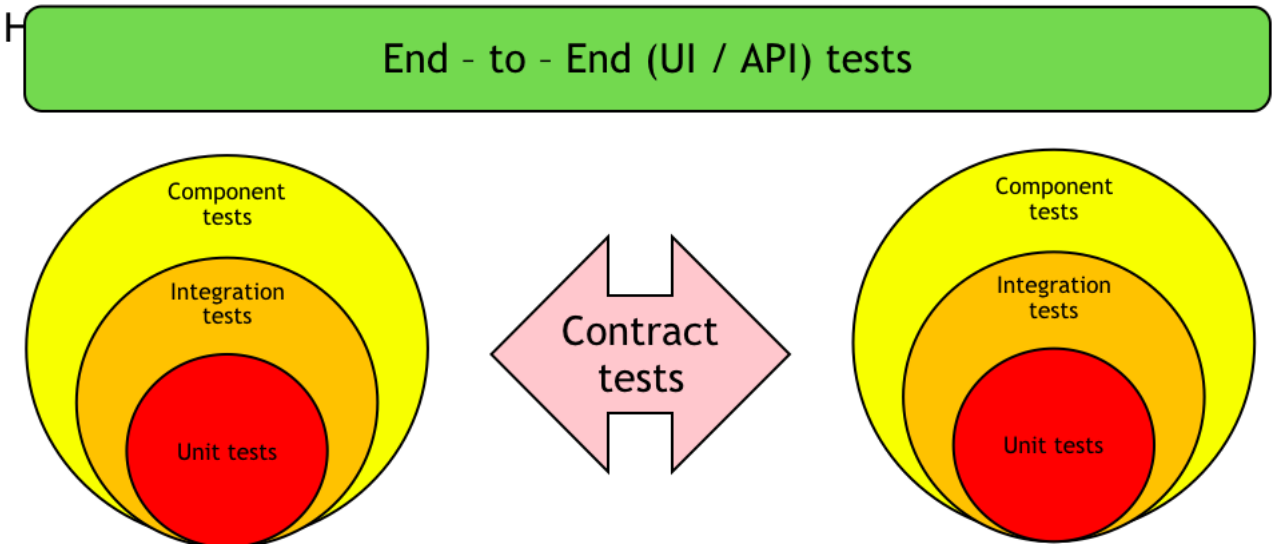

Рисунок 2.8 – Транформована схема рівнів тестування для мікросервісної архітектури

В схемі у доповнення до класичних її частин новим елементом з'явилась ланка контрактного тестування. І таке тестування контрактів використовується для перевірки інтеграції між службами.

Тести інтерфейсу та API також є частиною тестової схеми, але їх кількість мінімізована і повинна охоплювати лише критичні бізнес-потоки програми, що тестується.

Крім того, навіть такий модифікований підхід до тестування не буде надійним без ручного тестування додатків. Це може бути попереднє дослідження, тестування UX та ін. Тобто розробники-користувачі мікросервісної архітектури не повинні нехтувати цим класичним типом тестування.

2.5. Висновки по розділу 2

В рамках вирішення задачі верифікації інтерфейсів у системах з архітектурою за принципом слабого зв'язування окремих компонентів системи у другому розділі виконано наступні завдання:

- проаналізовано принципи взаємодії компонентів у мікросервісній архітектурі як окремих незалежних модулів з власним API;
- досліджено устрій інтерфейсів за технологією RESTful;
- визначено сутність веріфікації інтерфейсу у мікросервісній архітектурі;
- досліджено метод контрактного тестування, його особливості та відмінності;
- запропоновано побудувати новий підхід до виконання верифікації інтерфейсів слабо зв'язаних програмних компонентів, що базується на контрактному тестуванні.

3. Засоби реалізації програмної системи

3.1. Побудова контрактів мікросервісів на основі протоколу http

HTTP - це протокол, який дозволяє отримувати різні ресурси, наприклад документи HTML. HTTP є основою для обміну даними в Інтернеті. HTTP - це протокол для клієнт-серверної взаємодії, що означає ініціювання запитів до сервера одержувачем, зазвичай через веб-браузер. Отриманий підсумковий документ реконструюється з різних документів, що додаються, наприклад, з тексту, отриманого окремо, опису структури документу, зображень, відеофайлів, сценаріїв тощо (рис 3.1) [9].

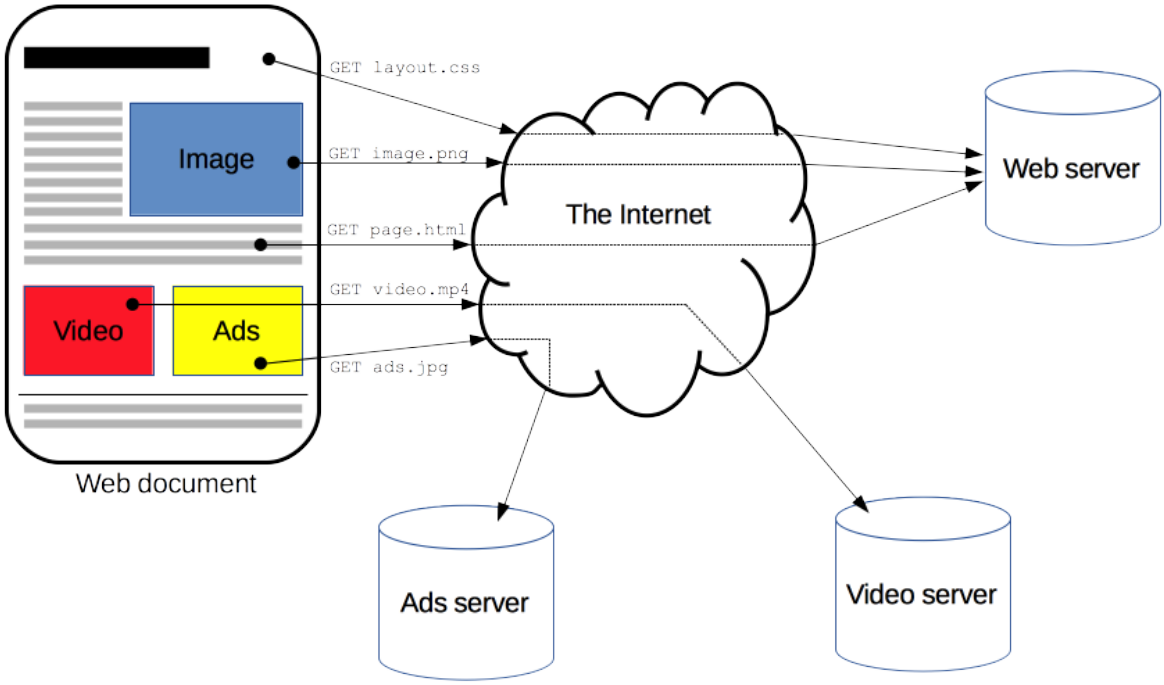

Рисунок 3.1 –Композиція веб-документу

Клієнти та сервери обмінюються даними, обмінюючись окремими повідомленнями (на відміну від потоку даних). Повідомлення, надіслані клієнтом, зазвичай через веб-браузер, називаються запитами (рис 3.2) та повідомлення, що надіслані сервером у відповідь, називаються відповідями (рис 3.3).

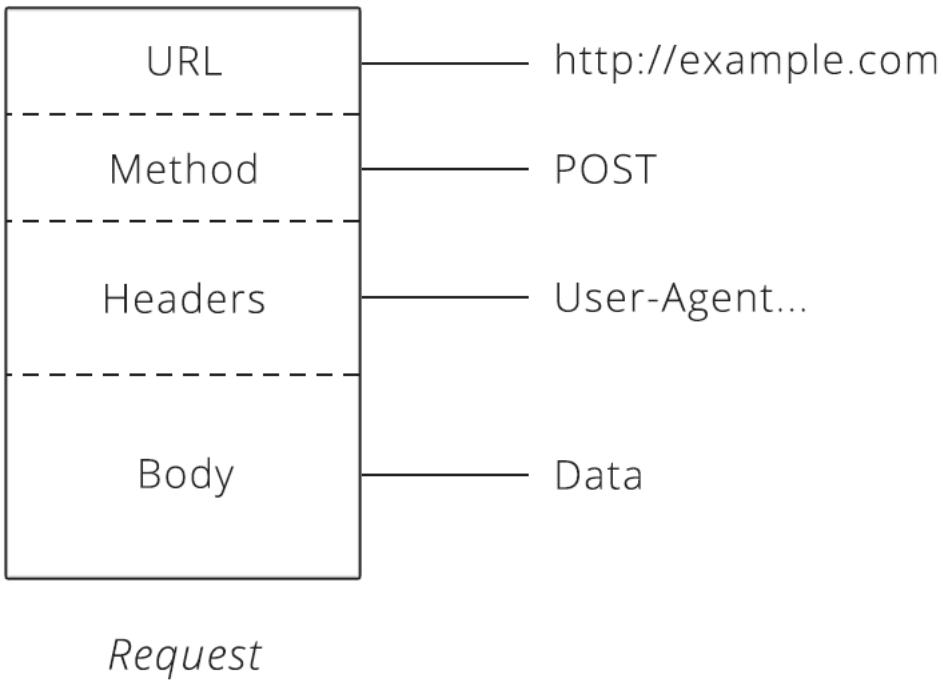

Рисунок 3.2 – Загальна структура запиту

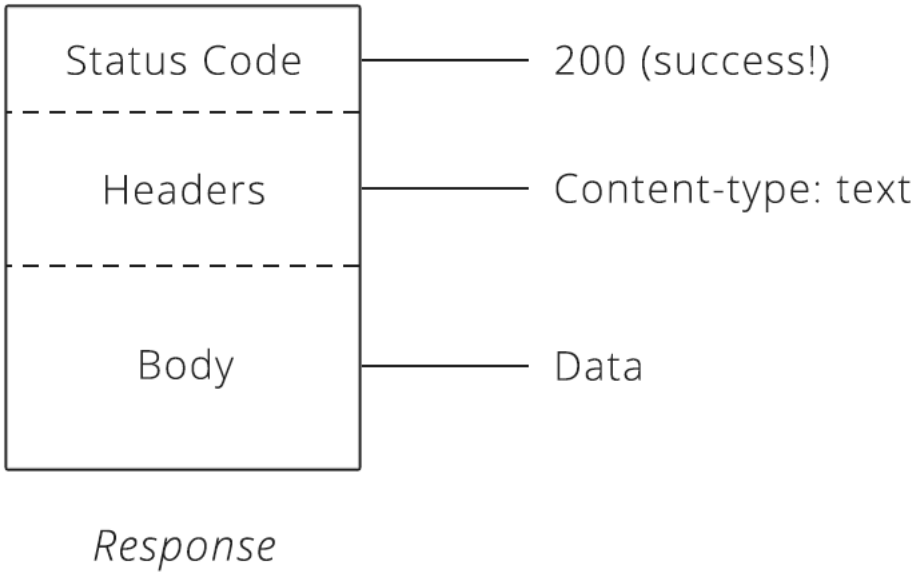

Рисунок 3.3 – Загальна структура відповіді

Розроблений на початку 1990-х років, HTTP – це розширюваний протокол, який розвивався з часом. Це протокол рівня додатків, який надсилається через TCP або TCP-з'єднання за допомогою шифрування TLS, хоча теоретично можна використовувати будь-який надійний транспортний протокол. Завдяки своїй розширюваності він використовується не тільки для вилучення гіпертекстових

документів, але також для зображень та відео, а також для розміщення даних на серверах, як у результатах HTML-форм. HTTP також може використовуватися для вилучення окремих документів для оновлення веб-сторінок за вимогою.

HTTP – це протокол клієнт-серверної взаємодії: запити надсилаються одним об'єктом, агентом користувача (або проксі-сервером від його імені). У більшості випадків агентом користувача є веб-браузер, але це може бути будь-що, наприклад, робот, який сканує Інтернет для заповнення та підтримки індексу пошукової системи [10].

Кожен індивідуальний запит надсилається на сервер, який буде обробляти його та надавати відповідь. Між цим запитом та відповіддю існує безліч об'єктів, спільно позначених як проксі, які виконують різні операції і діють, наприклад, як шлюзи або кеш-пам'ять (рис 3.4).

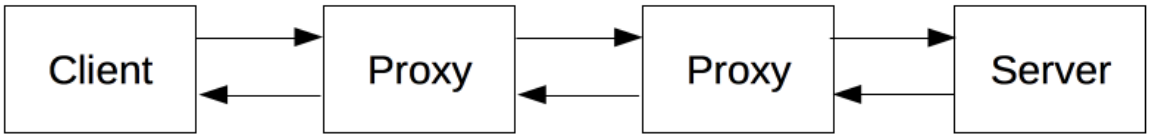

Рисунок 3.4 –Ланцюг ланок клієнт-серверної взаємодії

Фактично, між браузером і сервером обробляється запит більшою кількістю пристроїв: є маршрутизатори, модеми та багато іншого. Завдяки багаторівневому веб-дизайну вони приховані в мережі та транспортних шарах. HTTP знаходиться на верхній частині прикладного рівня. Хоча є важливим діагностувати мережеві проблеми, та основні шари мережевої моделі не мають співвідношення до опису HTTP.

Агентом користувача є будь-який інструмент, який діє від імені користувача. Ця роль в основному виконується веб-браузером; кілька виключень - це програми, які інженери та веб-розробники використовують для налагодження прикладних програм.

Браузер завжди є об'єктом, який ініціює запит. Він ніколи не є сервером (але останнім часом створено та може використовуватись певні механізми для імітації повідомлень, ініційованих сервером).

Для відображення веб-сторінки браузер надсилає оригінальний запит, щоб отримати HTML-документ з цієї сторінки. Потім він аналізує цей файл, вибираючи додаткові запити, що відповідають сценаріям виконання, відображену інформацію про макет (CSS) та підресурси, що містяться на сторінці (як правило, зображення та відео). Потім веб-браузер об'єднує ці ресурси, щоб надати користувачеві повний документ веб-сторінки. Сценарії, виконані браузером, можуть отримати більше ресурсів на більш пізньому етапі, тоді веб-браузер оновлює відповідну веб-сторінку.

Веб-сторінка - це гіпертекстовий документ. Це означає, що деякі частини тексту, що відображається - це посилання, які можуть бути активовані (зазвичай гіпертекстовим переходом), щоб отримати нову веб-сторінку, яка дозволяє користувачеві здійснювати навігацію по Інтернету. Браузер переводить ці інструкції в HTTP-запити та додатково інтерпретує HTTP-відповіді, щоб надати користувачеві чітку відповідь.

На протилежній стороні каналу зв'язку очікує на запити інший учасник веб-взаємодії - сервер, який обслуговує документ за вимогою клієнта. Сервер представляється як єдиний монолітний ресурс, але фізично це може бути сукупність серверів, що обмінюються навантаженням (балансування навантаження) або складний комплекс програмного забезпечення для опитування інших комп'ютерів (наприклад, кеш-пам'яті, серверу БД, серверів електронної комерції та ін.), повністю або частково генеруючи документ за вимогою.

Сервер не обов'язково є однією машиною, але декілька серверів можуть розміщуватися на одній машині. За допомогою заголовка HTTP/1.1 та заголовка вони можуть навіть мати таку ж IP-адресу.

Численні комп'ютери та машини надсилають HTTP-повідомлення між веб-браузером та сервером. Через шарувату структуру веб-стеку більшість з них працюють на транспортному, мережевому або фізичному рівнях, стають прозорими на рівні HTTP і потенційно можуть суттєво вплинути на продуктивність. Ті, хто працює на рівні застосунків, зазвичай називають проксі.

Вони можуть бути прозорими чи ні (змінюючи запити, що проходять через них), і можуть виконувати багато функцій:

- кешування (кеш може бути загальнодоступним або приватним, як кеш-пам'ять браузера);
- фільтрація (наприклад, перевірка антивірусом, батьківський контроль та ін.);
- балансування навантаження (для того, щоб кілька серверів могли обробляти різні запити);
- автентифікація (для контролю доступу до різних ресурсів);
- реєстрація (що дозволяє зберігати історичну інформацію).

Навіть з більшою складністю, введеною в HTTP/2 шляхом інкапсулювання HTTP-повідомлень у фрейми, HTTP, як правило, призначений для простоти та зручності у читанні. Фахівці можуть читати та розуміти HTTP-повідомлення, що спрощує тестування для розробників та зменшує складність для початківців.

Введені в HTTP/1.0, HTTP заголовки зробили цей протокол легким для розширення та експериментів. Нова функціональність може бути впроваджена простою угодою між клієнтом і сервером про семантику нового заголовка.

HTTP вважається протоколом без пам’яті, тому що не має стану: між двома запитами, які виконуються послідовно через одне з'єднання, немає зв'язку, отже передачі даних. Це може негайно стати проблемою для користувачів, які намагаються послідовно взаємодіяти з певними сторінками, наприклад, використовуючи кошики для електронної комерції. Але в той час як сам HTTP-сервер не має стану, HTTP-файли cookie дозволяють використовувати сеанси, що відрізняються від стану. Використовуючи розширення заголовків, файли cookie HTTP додаються до робочого процесу, що дозволяє створювати сесії для кожного HTTP-запиту таким чином, щоб вони мали однаковий контекст або такий самий стан.

З'єднання контролюється на транспортному рівні, і, отже, принципово виходить за межі HTTP. Хоча HTTP не вимагає, щоб основний транспортний протокол базувався на з'єднанні; лише вимагаючи, щоб він був надійним або не втратив повідомлення (тому, як мінімум, представив помилку). Серед двох найбільш поширених транспортних протоколів в Інтернеті, TCP є надійним, але UDP - не є. HTTP згодом почав спиратися на стандарт TCP, який базується на з'єднанні, хоча з'єднання не завжди є обов'язковим.

HTTP/1.0 відкрив TCP-з'єднання для кожного обміну запитів/відповідей, ввівши два основних недоліки: для відкриття з'єднання потрібні кілька циклів обміну повідомленнями, а, отже, повільно, але обміни стають більш ефективними, коли кілька повідомлень надсилаються і надсилаються регулярно: теплі з'єднання більш ефективні, ніж холодні.

Щоб усунути ці недоліки, HTTP/1.1 запровадив конвеєрне з'єднання (яке виявилося важким для реалізації) та постійні з'єднання: базовий TCP-зв'язок може бути частково керованим за допомогою заголовка з'єднання. HTTP/2 отримав подальший розвиток: повідомлення стали складнішими через одне підключення, що допомагає підтримувати між ними зв'язок, що є більш ефективним.

В даний час проводяться експерименти з розробки кращого транспортного протоколу, більш придатного для HTTP. Наприклад, Google експериментує з QUIC, заснованим на UDP, для забезпечення більш надійного та ефективного транспортного протоколу.

Цей розширюваний характер HTTP з часом дозволив би забезпечити більший контроль та функціональність в Інтернеті. Методи кешування або аутентифікації були функціями, які оброблялися на початку історії HTTP. Здатність ослабити обмеження походження, навпаки, було додано лише в 2010-х роках.

Основний список загальних функцій, керованих за допомогою HTTP:

- кеш-пам'ять - які документи кешуються, можна вкузувати через HTTP. Сервер може вказати проксі та клієнти, які кешуються, і на скільки часу. Клієнт може зконфігурувати проміжні проксі-сервери для ігнорування збереженого документа.

- Пом’якшення обмежень джерела - щоб запобігти відстеженню та іншим порушенням конфіденційності, веб-браузери забезпечують чітке розділення веб-сайтів. Лише сторінки спільного походження можуть отримати доступ до всієї інформації на веб-сторінці. Незважаючи на те, що це обмеження є серйозним навантаженням, заголовки HTTP можуть послабити цей жорсткий поділ на стороні сервера, що дозволить перетворити документ на різноманітну інформацію з різних доменів (може навіть бути з причин безпеки).
- Аутентифікація - деякі сторінки можуть бути захищені, тому доступ до них можуть отримати лише певні користувачі. Базова аутентифікація може бути надана HTTP, використовуючи WWW-Authenticate та аналогічні заголовки, або настроюючи певний сеанс за допомогою файлів cookie HTTP.
- Проксі і тунелювання - сервери та/або клієнти часто розміщуються в інтрамережі та приховують свою справжню IP-адресу від інших. Потім HTTP-запити проходять через проксі, щоб подолати цей мережевий бар'єр. Не всі проксі - це проксі HTTP. Наприклад, протокол SOCKS працює на більш низькому рівні. Інші, такі як ftp, можуть оброблятися цими проксі-серверами.
- Сесії - використання файлів cookie HTTP дозволяє пов'язати запити зі станом сервера. Це створює сеанси, навіть якщо базовий HTTP є протоколом без статусу. Це корисно не тільки для кошиків для електронної торгівлі, але і для будь-якого сайту, який дозволяє зконфігурувати виведення інформації користувачем.

Коли клієнту потрібно спілкування з сервером, який є кінцевим або проміжним проксі-сервером, він виконує наступні кроки:

1. Відкрити з'єднання TCP: TCP-з'єднання буде використовуватися для надсилання одного або кількох запитів і отримання відповіді. Клієнт

може відкрити нове з'єднання, повторно використовувати існуюче з'єднання або відкрити декілька з'єднань TCP із серверами.

2. Надіслати HTTP-повідомлення: HTTP-повідомлення (до HTTP/2) читаються людьми. У HTTP/2 ці прості повідомлення оформляються, що робить їх неможливим для безпосереднього читання, але принцип залишається незмінним.

```
GET/HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

3. Прочитати відповідь, надіслану сервером:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html...
```

4. Закрити або повторно використати з'єднання для подальших запитів.

Якщо HTTP комунікація активована, можливо надіслати кілька запитів, не чекаючи повної відповіді на перший запит. Проект HTTP виявився складним у існуючих мережах, де старі програмні продукти співіснують з сучасними версіями. Конфігурація HTTP була замінена в HTTP/2 з більш надійними складними запитами в кадрі.

HTTP/1.1 та попередні повідомлення HTTP мають форму тексту, тому доступні для читання людині-спеціалісту. У HTTP/2 ці повідомлення вбудовані в нову бінарну структуру - кадр, який дозволяє оптимізувати, наприклад, стиснення

заголовків та мультиплексування. Навіть якщо в цій версії HTTP відправляється лише частина оригінального HTTP-повідомлення, семантика кожного повідомлення залишається незмінною, і клієнт відновлює (фактично) оригінальний запит HTTP/1.1. Тому корисно розуміти повідомлення HTTP/2 у форматі HTTP/1.1.

Є два типи HTTP-повідомлень, запити і відповіді, кожен з яких має свій власний формат.

Запит складається з наступних елементів (рис 3.5):

- HTTP-метод це зазвичай дієслово типу GET, POST або іменник типу OPTIONS або HEAD, який визначає операцію, яку хоче виконати клієнт. Як правило, клієнт хоче отримати ресурс (використовуючи GET) або опублікувати значення HTML-форми (використовуючи POST), хоча в інших випадках може знадобитися більше операцій.
- Шлях до ресурсу для вилучення; URL-адреса ресурсу витягується з елементів, які є очевидними з контексту.
- Версія протоколу HTTP.
- Додаткові заголовки, які передають додаткову інформацію для серверів.
- Тіло (для деяких методів, таких як POST) подібне до тих, що містяться у відповідях, що містять опублікований ресурс.

Відповіді складаються з наступних елементів (рис 3.6):

- Версія протоколу HTTP, яку вони дотримуються.
- Код стану, який вказує, чи був запит успішним, чи ні, і чому.
- Повідомлення про стан, неавторитетні короткі описи коду стану.
- Заголовки HTTP.
- Тіло (необов'язково), що містить завантажений ресурс.

Рисунок 3.5– Приклад http-повідомлення запиту

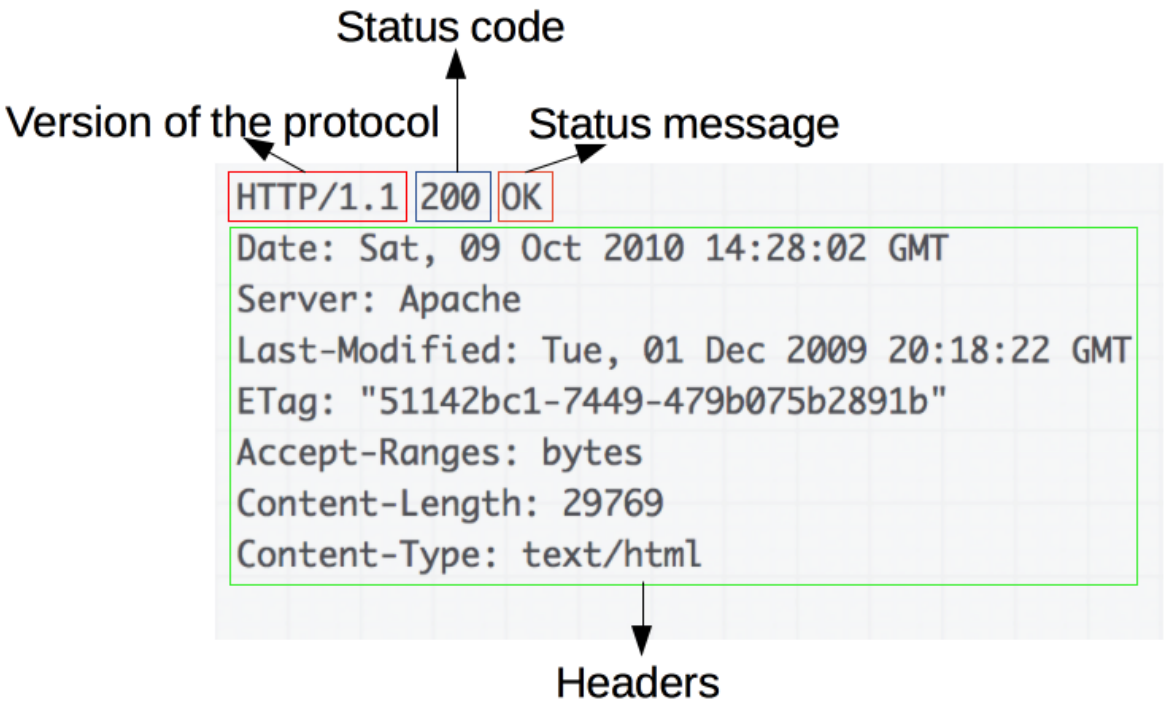

Рисунок 3.6 – Приклад http-повідомлення відповіді

Найпоширеніший API, що базується на HTTP, - це XMLHttpRequest API, який можна використовувати для обміну даними між агентом користувача та сервером.

Інший API, який надсилається на сервер за подіями, є службою в одну сторону, яка дозволяє серверу передавати події клієнту за допомогою HTTP як

механізму транспортування. Використовуючи інтерфейс EventSource, клієнт відкриває зв'язок та встановлює обробники подій. Клієнт автоматично перетворює повідомлення, що потрапляють у потік HTTP, у відповідні об'єкти події, передаючи їх обробникам подій, які були зареєстровані для типу події (якщо вони відомі) або обробника подій надісланого повідомлення, якщо обробника подій певного типу не було встановлено.

HTTP - розширюваний протокол, який простий у використанні. Структура клієнт-сервер, у поєднанні з можливістю простого додавання заголовків, дозволяє HTTP розширюватися разом із сучасними можливостями Інтернету.

Хоча HTTP/2 додає певну складність, вставляючи HTTP-повідомлення в фрейми для підвищення продуктивності, основна структура повідомлень залишається незмінною з HTTP/1.0. Сесійний потік залишається простим, дозволяючи його досліджувати та налагоджувати, використовуючи простий монітор HTTP-повідомлень.

3.2. Платформа розробки та виконання .NET Core

.NET Core має такі характеристики [1]:

- Крос-платформеність: працює в операційних системах Windows, MacOS та Linux.
- Послідовний у всіх видах архітектури: керування кодом з однаковою поведінкою в декількох архітектурах, включаючи x64, x86 та ARM.
- Інструменти командного рядка: включає в себе прості у використанні інструменти командного рядка, які використовуються для локальної розробки та сценаріїв безперервної інтеграції.
- Гнучке розгортання: може бути включено у ваш приклад або встановлено поруч на стороні користувача або комп'ютера. Можна використовувати з контейнерами Docker.

- Сумісність:. NET Core сумісний з .NET Framework, Xamarin і Mono, через .NET Standard.
- Відкритий вихідний код: платформа .NET Core відкрита, використовуючи ліцензії MIT та Apache 2. NET Core є проектом .NET Foundation.
- Підтримується корпорацією Майкрософт: .NET Core підтримується корпорацією Майкрософт за підтримки Core .NET Support.

Мови C #, Visual Basic та F # можуть бути використані для написання програм і бібліотек для .NET Core. Ці мови можуть бути інтегровані у ваші улюблені текстові редактори та IDE, зокрема Visual Studio, Visual Studio Code, Sublime Text та Vim. Ця інтеграція частково забезпечується хорошими людьми з проектів OmniSharp та Ionide.

NET Core виставляє API для багатьох сценаріїв, деякі з яких слідують:

- Примітивні типи, такі як bool та int.
- Колекції, такі як System.Collections.Generic.List <T> та System.Collections.Generic.Dictionary <TKey, TValue>.
- Типи утилітних програм, такі як System.Net.Http.HttpClient та System.IO.FileStream.
- Типи даних, такі як System.Data.DataSet, і DbSet.
- Висока продуктивність типів, таких як System.Numerics.Vector.

.NET Core забезпечує сумісність із .NET Framework та Mono API, реалізуючи специфікацію стандарту .NET.

Багаточисельні фреймворки побудовані на.NET Core:

- ASP.NET Core
- Windows 10 Universal Windows Platform (UWP)
- Тизен
- NET Core складається з наступних частин:
- Керування .NET Core, яке забезпечує системні типи, збирання збірок, смітник, нативну взаємодію та інші базові послуги. Корпоративні бібліотеки .NET

Core забезпечують примітивні типи даних, типи компонентів додатків та основні утиліти.

- Середовище виконання ASP.NET, яке створює основу для побудови сучасних хмарних інтернет-підключених програм, таких як веб-додатки, додатки IoT та мобільний бекенд.
- Інструменти .NET Core CLI та компілятори мови (Roslyn та F #), які дозволяють досвіду розробника .NET Core.
- Інструмент "dotnet", який використовується для запуску додатків .NET Core і інструментів CLI. Він вибирає час виконання та встановлює час виконання, забезпечує політику збирання навантажень та запускає додатки та інструменти.

Ці компоненти поширюються таким чином:

- .NET Core Runtime - включає в себе бібліотеки середовища .NET Core і бібліотеки.
- ASP.NET Core Runtime - включає в себе основні бібліотеки ASP.NET Core і .NET Core і бібліотеки.
- .NET Core SDK - включає в себе .NET CLI Tools, ASP.NET Core і Runtime і Core .NET.

3.4. NuGet для підвантаження програмного модулю фреймворку для тестування

Важливим моментом для розвитку будь-якої сучасної платформи є забезпечення механізму (рис 3.7), за допомогою якого розробники можуть створювати, поширювати та використовувати корисний код. Часто такий код об'єднується в "пакети", які містять зібраний код (у форматі DLL) разом з вмістом, що є необхідним для цих пакетів [8].

Для. NET (включаючи .NET Core) таким механізмом є онлайн-ресурс NuGet, що підтримується Microsoft. Він забезпечує створення, доступність та поширення пакетів для .NET та надає інструменти для цього.

Тобто, NuGet - це єдиний ZIP-файл .nupkg, який містить зібраний код (DLL) та інші файли, пов'язані з цим кодом, і базову документацію, яка включає в себе таку інформацію, як версія пакету. Розробники використовують програмний код та загальнодоступні пакети для доступу до публічних чи приватних серверів хостінгу. Вони впроваджують функціональні пакети в коді проекту. Тоді як NuGet обробляє всі проміжні деталі.

Таким чином, NuGet підтримує приватний хостинг, а також загальнодоступний хостинг Nuget.org, де є можливість розмістити пакети виключно для вашої організації або робочої групи. Також є можливість використовувати загальнодоступні NuGet пакети. Тобто пакет NuGet є спільним каналом, але не вимагає або не передбачає використання будь-яких конкретних засобів широкого використання.

Як публічний хост, NuGet підтримує центральне сховище з понад 100 000 унікальних пакетів на nuget.org. Ці пакети використовуються мільйонами розробників .NET/.NET Core щодня. NuGet також дозволяє розміщувати пакети в приватному режимі у хмарному сховищі (наприклад, у DevOps Azure), в приватній мережі або навіть просто в локальній файловій системі. Таким чином, ці пакети доступні лише тим розробникам, які мають доступ до вузла, що надає можливість зробити пакети доступними для конкретної групи споживачів. Усі деталі розгортання можна вказати під час завантаження пакету в NuGet. За допомогою параметрів конфігурації також можливо визначити, які вузли можна отримати з будь-якого комп'ютера, тим самим гарантуючи, що пакети будуть отримуватися з певних джерел, а не з публічного сховища, наприклад nuget.org.

Незалежно від його характеру, хост служить точкою контакту між розробниками та споживачами пакетів. Розробники створюють корисні NuGet-пакети та публікують їх на хості. Тоді споживачі шукають корисні та сумісні пакети на доступних ресурсах, завантажуючи і включаючи ці пакети в свої проекти.

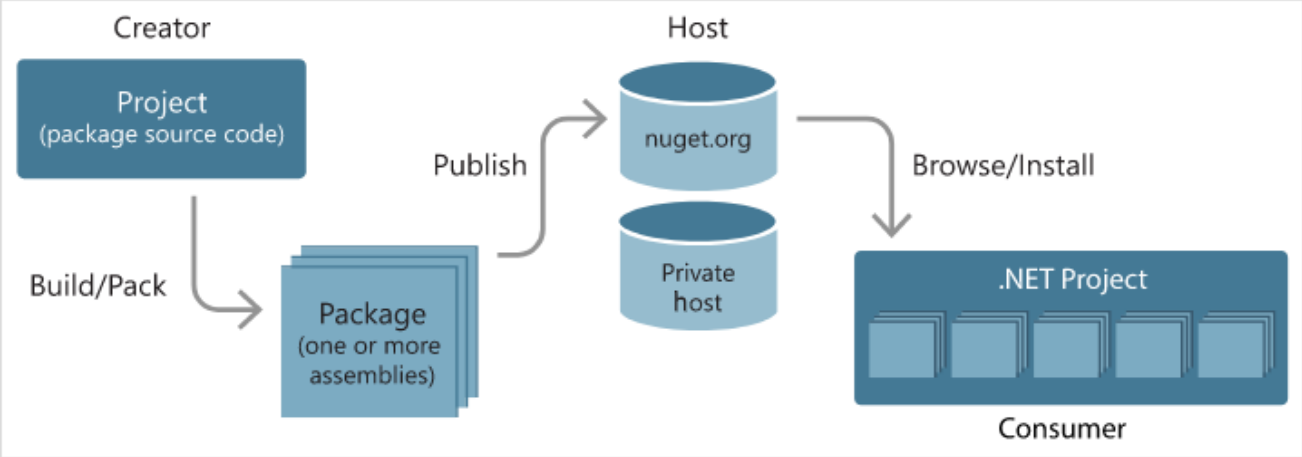

Рисунок – 3.7 Шлях бібліотеки від створення до використання

«Сумісний» пакет означає, що він містить збірки, зібрані принаймні для однієї цільової платформи .NET, яка сумісна з цільовою платформою споживача проекту. Розробники можуть створювати пакети, що відповідають конкретній структурі, як і елементи керування UWP, або вони можуть підтримувати більш широкий діапазон цілей. Для максимальної сумісності пакетів розробники використовують .NET Standard, який можуть використовувати всі проекти .NET та .NET Core. Це найефективніший інструмент як для розробників, так і для споживачів, оскільки єдиний пакет (який зазвичай містить одну збірку) працює для всіх споживачів проектів.

Розробники пакетів, які вимагають API за межами стандарту .NET, з іншого боку створюють окремі збірки для різних цільових платформ, які вони хочуть підтримувати, і включати всі ці збірки в один пакет. Коли споживач встановлює такий пакет, NuGet вилучає лише ті збірки, які потрібні для проекту. Це зменшує розмір пакету в кінцевому застосунку та/або збірках, створених цим проектом. Звичайно, багатоцільовий пакет значно складніший у підтримці для його розробника.

Крім підтримки хостингу, NuGet також надає безліч інструментів:

- Nuget.exe CLI - надає всі можливості NuGet, деякі команди, які важливі лише для розробників пакетів, деякі застосовуються лише для споживачів, а інші для обох. Наприклад, розробники пакетів використовують команду Nuget Pack для створення пакетів із різних агрегатів та файлів, споживачі пакетів використовують

команду Nuget install, щоб включити пакети в папку проекту, і всі вони використовують конфігурацію Nuget для налаштування NuGet.

- Dotnet CLI - надає певні функції NuGet CLI у зв’язці з інструментами .NET Core. Як і з NuGet CLI, інтерфейс командного рядка dotnet не взаємодіє з проектами Visual Studio [13].
- Package Manager Console - надає команди PowerShell для встановлення та управління пакетами в програмах Visual Studio.
- Package Manager UI - користувальницький інтерфейс диспетчера пакетів програм Visual Studio у інтерфейсі користувача Windows для встановлення та керування пакунками у програмах Visual Studio.
- Manage NuGet UI – надає функції для встановлення та керування пакетів у програмах Visual Studio для Mac.
- MSBuild – Надає функції створення, використовування, пакування та відновлення пакетів.

Інструменти NuGet запускаються на платформі, на якій працюють розробники.

Розробники пакетів, як правило, є також споживачами, оскільки вони покладаються на функціональність, що існує в інших пакетах NuGet. Тому створені пакети зазвичай використовуються в інших пакетах NuGet. Звичайно, вони можуть мати залежності від інших.

Вміння легко покладатися на роботу інших людей є однією з найпотужніших функцій системи керування пакетами. Відповідно більшість того, що робить NuGet, полягає в керуванні деревом залежностей або "розкладом" від імені проекту. Тобто програміст опікується лише тими пакетами, які безпосередньо використовує у власному проекті. Якщо будь-який з цих пакетів самостійно використовує інші пакети (які, в свою чергу, можуть використовувати інші), NuGet буде дбати про всі ці нижчі рівні залежностей.

Наступний рисунок (рис 3.8) показує проект, який залежить від п'яти пакетів, які, у свою чергу, залежать від ряду інших.

Наприклад, є три різних споживача пакета B, і кожен користувач може також вказати свою версію для цього пакета. Це поширена практика, особливо для широко використовуваних пакетів. І NuGet «вирішує» цю колізію, щоб точно визначити, яка версія пакету B задовольняє всіх споживачів. Потім NuGet виконує те ж саме для всіх інших пакетів, незалежно від того, наскільки глибокий граф залежності.

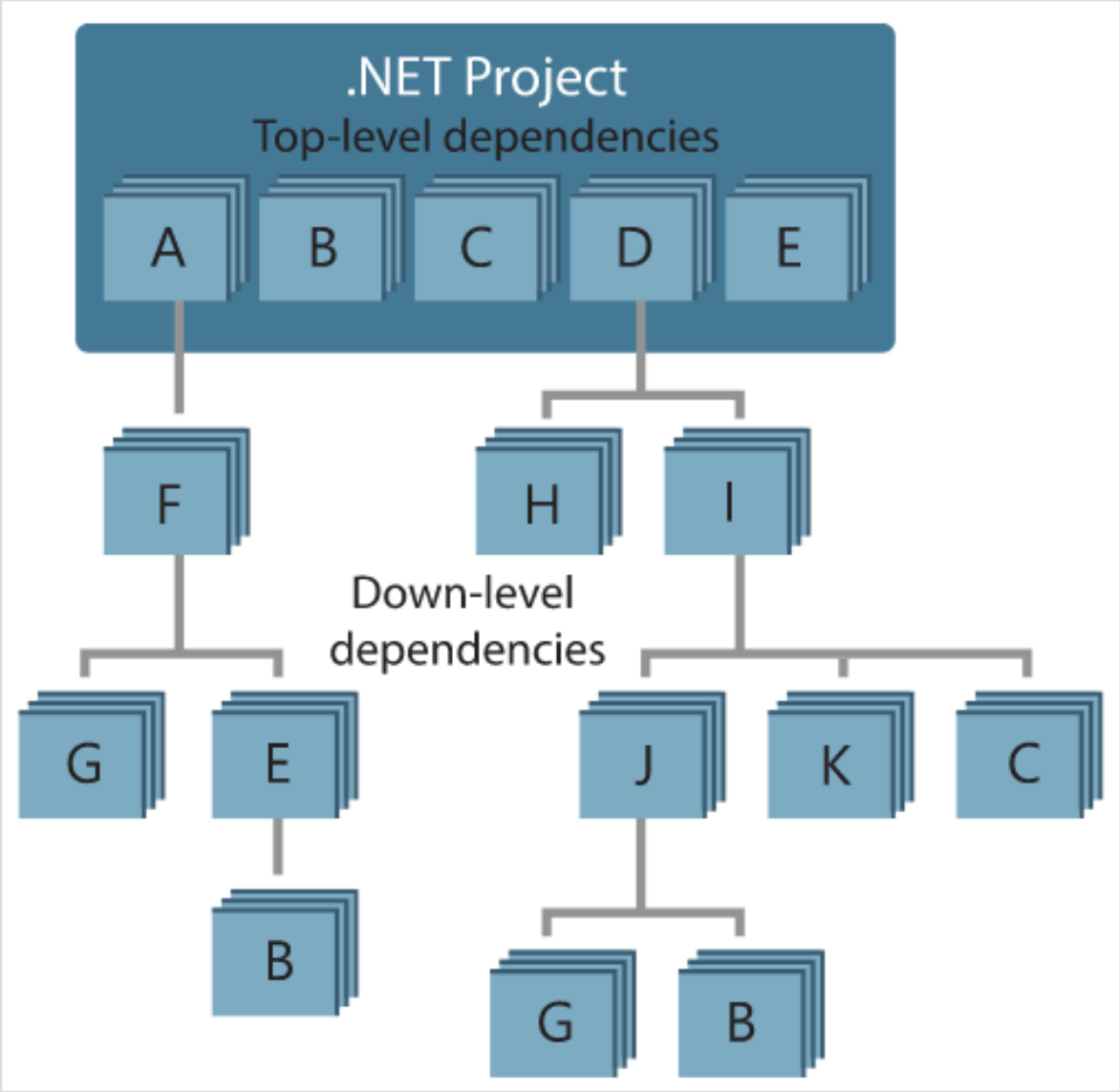

Рисунок 3.8 – Приклад залежностей між пакетами

Оскільки проекти можуть легко переміщатися між комп'ютерами розробників, сховищами керування вихідним кодом, створеними серверами тощо., це дуже непрактично підтримувати дублікати пакетів NuGet, що безпосередньо пов'язані з проектом. Це призведе до того, що кожна копія проекту надмірно

збільшується за розміром. І дуже важко оновити бінарні файли пакету згідно нової версії, оскільки оновлення повинні бути застосовані до всіх копій проекту.

Натомість NuGet підтримує простий список посилань на пакети, від яких залежить проект, включаючи залежності як від верхнього, так і до нижнього рівнів. Тобто, коли встановлюється пакет з хостінгу на проект, NuGet записує ідентифікатор пакета та номер версії до списку посилань. (якщо видалити пакет, він видаляється зі списку.) Після цього програма NuGet забезпечує відновлення всіх пакетів, на які є посилання, за запитом (рис 3.9).

Маючи лише список посилань, NuGet може перевстановити, тобто відновити всі ці пакети з публічних і/або приватних хостів у будь-який час. Коли потрібно передати проект до системи керування версіями або поширити його іншим способом, необхідно включити лише список посилань, за цим списком і NuGet пізніше підключає всі бінарні файли пакету.

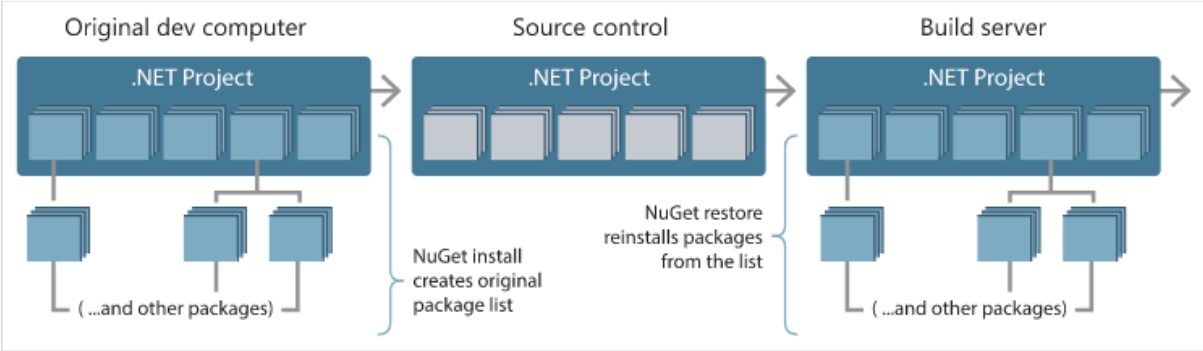

Рисунок 3.9 – Процес відновлення пакетів NuGet

Комп'ютер, який отримує проект, такий як сервер збірки, який отримує копію проекту як частину автоматизованої системи розгортання, надає запит до NuGet відновити потрібні залежності. Системи побудови, такі як DevOps Azure, забезпечують "NuGet Recovery" кроки з цією метою. Також коли розробники отримують копію проекту (наприклад, при клонуванні сховища вихідного коду), вони запускають таку команду, як відновлення nuget, відновлення dotnet або install-package, щоб отримати всі необхідні пакети. Зі свого боку, Visual Studio автоматично відновлює пакети під час створення проекту (за умови включення автоматичного відновлення) [7].

Очевидно, що задача NuGet-менеджеру полягає в тому, щоб зберегти цей список посилань від імені конкретного проекту та забезпечити засоби для ефективного відновлення (і оновлення) цих пакетів, на які посилається код проекту. Цей список підтримується в одному з двох форматів управління пакетами:

- packages.config: (NuGet 1.0+) XML-файл, що містить список всіх залежностей у проекті, включаючи залежності інших встановлених пакетів. Встановлені або відновлені пакети зберігаються в папці пакетів.
- PackageReference (або "посилання на пакети в файлах проекту") Підтримує список безпосередніх версій проектів у файлі проекту, тому окремого файлу не потрібно. Пов'язаний файл obj/project.assets.json створюється динамічно для керування загальним графіком залежностей пакетів, які використовує проект, а також усіма рівнями нижчого рівня. PackageReference завжди використовується в .NET Core проектах.

Формат управління пакетами, що використовується в будь-якому конкретному проекті, залежить від типу проекту та наявної версії NuGet (та/або Visual Studio). Щоб перевірити, який формат використовується, виконується пошук пакета package.config у кореневому проекті після встановлення першого пакету. Якщо цього файлу не існує, потрібно виконати пошук елементу <PackageReference> у файлі проекту.

Команди інтерфейсу командного рядка nuget.exe, такі як встановлення nuget, автоматично не додають пакет до списку посилань. Список оновлюється, коли пакет встановлюється за допомогою Visual Studio Package Manager (інтерфейс користувача або консоль) та використовується CLI dotnet.exe.

Щоб усі процеси працювали ефективно, NuGet виконує деякі приховані оптимізації. Зокрема, NuGet керує кешем пакетів і папкою глобальних пакетів для швидкого встановлення та відновлення. Кеш-пам'ять не завантажує пакет, який вже встановлений на комп'ютері. Папка глобальних пакетів дозволяє декільком проектам спільно використовувати один і той же встановлений пакет, тим самим зменшуючи загальний простір NuGet на комп'ютері. Папка кешу та глобальних

пакетів також дуже корисна для частого відновлення великої кількості пакетів, наприклад, на сервері збірки.

У рамках окремого проекту NuGet керує загальним графом залежностей, що знову включає в себе вирішення кількох посилань на різні версії одного пакунка. Нерідко проект має залежності від одного або декількох пакетів, які самі мають однакові залежності. Деякі з найбільш корисних пакетів на nuget.org використовуються багатьма іншими пакетами. Таким чином, у всьому графі залежності можливо мати десять різних посилань на різні версії того ж пакету. Щоб уникнути додавання декількох версій цього пакета до самої програми, NuGet визначає, яку версію можуть використовувати всі споживачі.

Крім того, NuGet підтримує всі специфікації, пов'язані з структурою пакетів (включаючи символи локалізації та конфігурацію) та способи їх посилання (включаючи діапазони версій та версії попереднього випуску). NuGet також надає різноманітні API для програмної роботи зі своїми службами, а також надає підтримку розробникам, які пишуть розширення на Visual Studio та шаблони проектів.

3.6. Середовище розробки Microsoft Visual Studio 2017

Microsoft Visual Studio - лінійка продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення і ряд інших інструментальних засобів. Дані продукти дозволяють розробляти як консольні додатки, так і додатки з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-додатки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, підтримуваних Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework і Silverlight [4].

Visual Studio включає в себе редактор вихідного коду з підтримкою технології IntelliSense і можливістю найпростішого рефакторінга коду. Вбудований

відладчик може працювати як відладчик рівня вихідного коду, так і відладчик машинного рівня. Решта вбудовуються інструменти включають в себе редактор форм для спрощення створення графічного інтерфейсу додатку, веб-редактор, дизайнер класів і дизайнер схеми бази даних. Visual Studio дозволяє створювати і підключати сторонні додатки (плагіни) для розширення функціональності практично на кожному рівні, включаючи додавання підтримки систем контролю версій вихідного коду (як, наприклад, Subversion і Visual SourceSafe), додавання нових наборів інструментів (наприклад, для редагування і візуального проектування коду на предметно-орієнтованих мовах програмування) або інструментів для інших аспектів процесу розробки програмного забезпечення (наприклад, клієнт Team Explorer для роботи з Team Foundation Server).

3.7. Веб-сервер Windows Server (IIS)

Роль веб-сервера (IIS) - забезпечити безпечну, легко керовану, модульну та розширювану платформу для надійного розміщення веб-вузлів, служб і додатків. Використання веб-сервера Служби IIS 8 забезпечує доступ до інформації користувачам в Інтернеті, інтрамережі і екстранет. Автоматизація призначеного для користувача інтерфейсу також дозволяє скриптам автоматичних тестів взаємодіяти з UI.

Переваги використання Служби IIS 8 для розгортання та хостингурозроблюваного web-додатку:

– максимізація рівня веб-безпеки завдяки скороченню обсягу сервера і автоматичної ізоляції додатків;

–просте розгортання та запуск веб-додатків ASP.NET, Classic ASP і PHP на одному сервері;

– ізоляція додатків шляхом унікальної ідентифікації робочих процесів і їх запуску в ізольованому середовищі за замовчуванням, що скорочує ризики небезпеки;

– просте додавання, видалення і заміна вбудованих компонентів IIS з налаштованими модулями, що відповідають потребам користувача;

– підвищення швидкості роботи веб-сайту за допомогою вбудованого динамічного кешування і розширеного стиснення.

3.8. Висновки по розділу 3

Для розробки підходу до верифікації інтерфейсів у системах з архітектурою за принципом слабого зв’язування окремих компонентів системи та засобів підтримки застосування розробленого підходу у третьому розділі виконано наступні завдання:

- визначено особливості протоколу http, що впливатимуть на склад та структуру контрактів мікросервісів;
- досліджено особливості застосування сучасної платформи розробки .Net Core;
- досліджено технологію компонування та завантаження програмного засобу підтримки написання коду з веріфікації інтерфейсів NuGet;
- розгорнуто для розробки та тестування програмних засобів веріфікації середовище MS Visual Studio 2017 та Web-сервер IIS.

4. Опис програмної реалізації

Розробка поділялася на 2 частини: написання прототипів компонентів з програмним інтерфейсом та побудова абстракції високого рівня для реалізації емуляторів, імітації основних викликів та опису контракту.

4.1. Розробка бібліотечного фреймворку як метод реалізації та застосування програмного модулю у функціональній верифікації

Визначення запропонованого у дипломній роботі підходу до функціональної верифікації інтерфейсів програмних модулів у системі слабо зв'язаних компонентів та детальний опис його сутності, алгоритму застосування ще не дає можливості почати його впровадження при розробці програмного забезпечення. Успішному швидкому застосуванню заважає ступінь технічної складності, вище якої оволодіння знаннями про підхід, достатніми для написання коду швидко та власноруч, без технічної підтримки чи допомоги, є настільки важким і працеємним, що ефективність та переваги застосування розробленого підходу зводяться до нуля.

Це не є недоліком чи підґрунтям рішення відмовитись від застосування запропонованого підходу, адже в галузі програмування та розробки програмного забезпечення з'явилось та використовується багато технологій та методів вирішення певних складних проблем і задач, які не можливо було б опанувати та впровадити без підтримки у вигляді бібліотек коду, шаблонів, фреймворків, тощо.

Розглянемо як приклад для наслідування дві достатньо складні технології та підходи, що використовуються широко програмістами для створення коду на платформі .Net.

Наприклад, сучасний підхід до розробки програмних систем, в яких використовуються бази даних. Класичний варіант написання коду, що звертається до реляційної бази даних, є використання класів ADO.Net, що інкапсулюють в собі

логіку використання провайдерів баз даних. Так, програміст створює об'єкти-з'єднання з базою даних, об'єкти-команди, що утримують в собі оператори на мові SQL, об'єкти виконання цих команд та читання і запису результатів команд. Тобто він має знати мову програмування, на якій розробляється програма, наприклад, java, C++ чи C# та мову роботи з джерелом даних, наприклад, T-SQL. Цю логіку роботи з БД із програмного коду запропоновано виконувати у вигляді ORM, задача якої приховати в собі деталі роботи з БД та оператори на мові джерела даних, а користувачеві надати можливість лише на мові програмування запрошувати дані та отримувати їх у вигляді об'єктів. Ідея ORM виявилась дуже цікавою для проектування, але складною для реалізації кожним програмістом для своїх продуктів окремо. Щоб спростити застосування такого підходу було вирішено розробити готову до використання бібліотеку як допомогу впровадити ORM у код швидко и якісно.

Одже, так з'явилась низка реалізацій ORM, готових до підключення до проектів, що працюють з базами даних. Для .Net був розроблений пакет Entity Framework. Даний пакет розміщено на всесвітньому онлайн-ресурсі Nuget.net і на сьогоднішній день кількість програмістів, що завантажили цей пакет до свого коду, дорівнюється мільйонам.

Суть даного пакету, що підтримує застосування ORM, полягає в тому, що він надає програмісту готові до наслідування базові класи, в яких інкапсулюється робота з провайдерами (драйверами) баз даних, перетворення команд мови програмування в команди T-SQL. Програмісту залишається лише пристосувати базовий функціонал до деякої предметної області. Для цього йому потрібно написати набір класів-сутностей на базі існуючих класів бібліотеки шляхом наслідування та виконати над ними операції згідно предметної області.

Налаштувати готовий ORM для своїх потреб значно зручніше, ніж розробляти весь код спочатку. Отже переваги та ефективність розробки та застосування такого підтримуючого бібліотечного коду як Entity Framework цілком очевидні.

Наведений випадок не є виключенням. За таким принципом програмісту надається засіб налаштування процесів авторизації та аутентифікації користувачів в програмних системах. Дана технологія ще складніша за попередню. Тому і бібліотека містить не тільки класи, що реалізують функціонал роботи з користувачами і групами користувачів, надавання їм дозволів і прав, але й код по використанню цих класів у певній послідовності (рис 4.1).

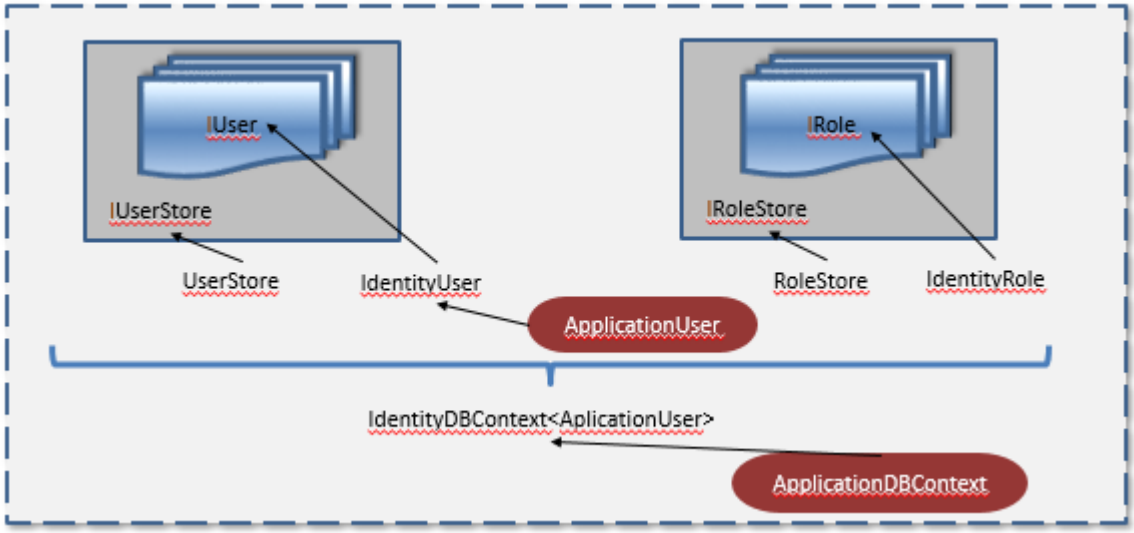

Рисунок 4.1 – система класів бібліотеки авторизації та аутентифікації

В середовищі розробки надається опція опитування програміста на бажання додати до проекту фунціонал авторизації та аутентифікації користувачів, а при підтвердженні в проекті розгортається структура системи аутентифікації на базі технології Identity. Програмісту залишаться налаштувати систему аутентифікації на потреби свого коду. Ці необхідні кроки значно простіше для розуміння, та виконати їх значно швидше. Отже, коло програмістів, яким стає доступним поріг складності використання системи аутентифікації, зростає в тисячі разів.

Тому для підтримки впровадження запропонованого в дипломній роботі підходу заплановано розробити бібліотеку класів, яка за принципом дії та призначенням буде виконувати функції, подібні до функцій пакетів Entity Framework та Identity, тобто сприяти застосуванню запропонованого у дипломній роботі підходу до функціональної верифікації інтерфейсів програмних модулів у

системі слабо зв’язаних компонентів програмістами при розробці програмних продуктів та їх компонентів.

Кожен компонент являє собою класичний мікросервіс з REST API.

4.2. Архітектура фреймворку контрактного тестування

Архітектура фреймворку має бути багаторівневою, але не занадто складною. Вона має розмежовувати рівні абстракції для відділення зв’язків компонетів та емуляторів від логіки.

Кожен окремий компонент за роллю має мати свої залежності, так як вони різняться за способами розгортання у віртуальних машинах.

За роллю компоненти поділяються на компоненти-постачальники (рис 4.3) та компоненти-споживачі (рис 4.2)

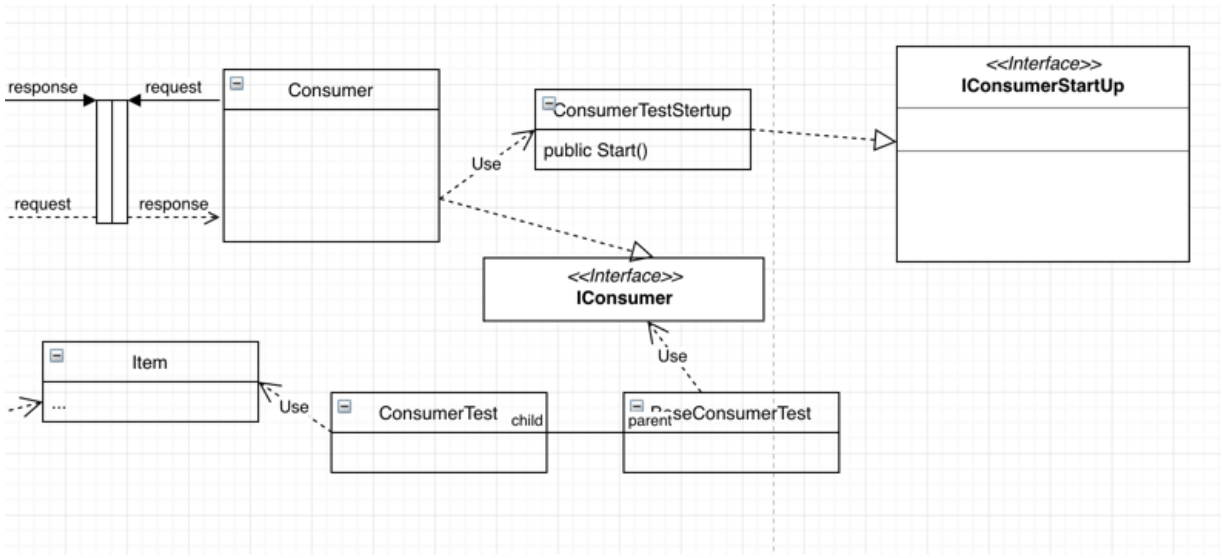

Рисунок 4.2 – Схема фреймворку для емулятора-споживача

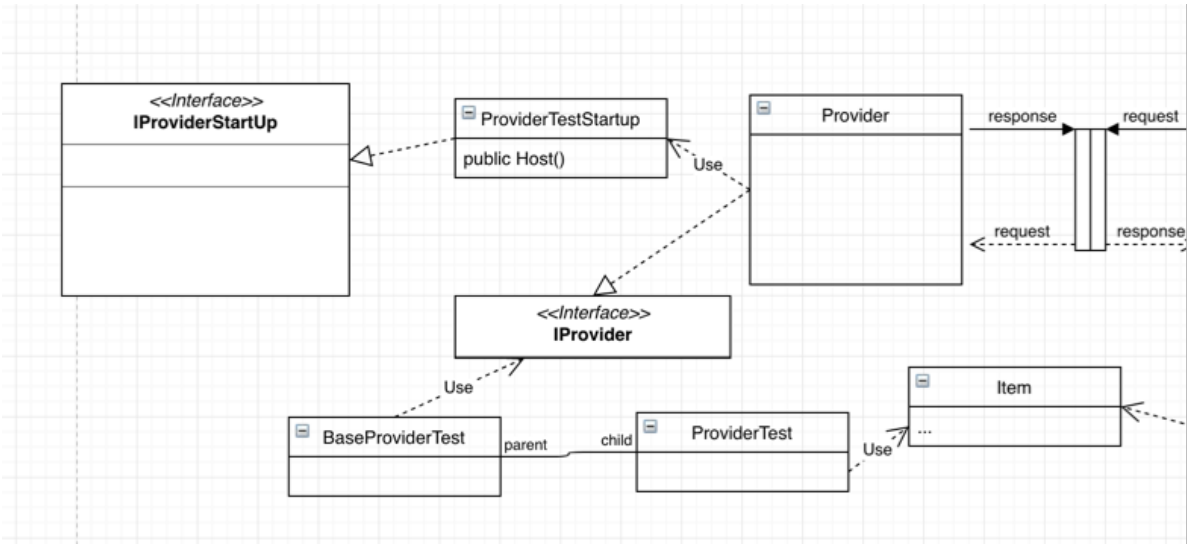

Рисунок 4.3 – Схема фреймворку для емулятора-постачальника

4.3. Прототип компонента

Так як емулятор це обгортка мікросервісу, то за основу береться структура та конфігурація мікросервісу. Запуск емулятора відбувається з файлу TestStartup, що успадковується від Startup мікросервісу. Конфігурація емулятора як мікросервіса також відбувається у файлі appsettings.json. Структура таких файлів одна й та сама. TestStartup має методи для емуляції мікросервісу та емуляції найпоширеніших запитів та оброки відповідей. Також емулятор має модель даних, з якою працює і яка представлена у вигляді класу, а поля моделі – у вигляді властивостей. Емулятор також має клас з методами-верифікаторами. У кожному з них оголошується контракт, мікросервіс та точка доступу. Далі виклик методу для хостінгу мікросервісу локально та хостінгу емулятора. Верифікація результату виконання запиту дозволяє порівняти очікуваний результат, прописаний у контракті, з реальним отриманим. Наступний крок верифікація структури відповіді згідно контракту та публікація контракту у сховище.

4.4. Рекомендації по розробці програмного забезпечення, яке планується тестувати фреймворком

Платформа: .Net Core

Версія: 2.1 і більше

Запуск мікросервісу відбувається з Startup файлу. Файл містить конструктор, який як параметр приймає конфігурацію мікросервісу, метод ConfigureServices(IServiceCollection services) метод, який використовується для впровадження залежностей зв'язків між іншими мікросервісами та метод Configure(IApplicationBuilder app, IHostingEnvironment env), який відповідає за налаштування каналів зв'язку. Усі три методи викликаються автоматично під час виконання.

Конфігурація мікросервісу прописується в appsettings.json файлі. Тут ми вказуємо рівні логування, дозволені хостінги та конфігураційні налаштування мікросервісів від яких даний компонент залежить. Модель даних, з якою працює даний мікросервіс, представлена у вигляді класу, а поля моделі – у вигляді властивостей. Іншою важливою частиною є класи контроллери. Такий клас включає в себе методи роботи зазвичай з окремою точкою доступу.

Так званий маршрут, шлях до точки доступу, вказаний у атрибуті над класом. Кожний частковий маршрут, до якого звертається конкретний метод, вказується у атрибуті над методом. Якщо мікросервіс повинен звертатися до іншого мікросервісу, то в такому випадку існує інтерфейс з сигнатурами методів мікросервісу-постачальника для їх викликів всередині.

Компонент-постачальник має мати наступну структуру (рис 4.4):

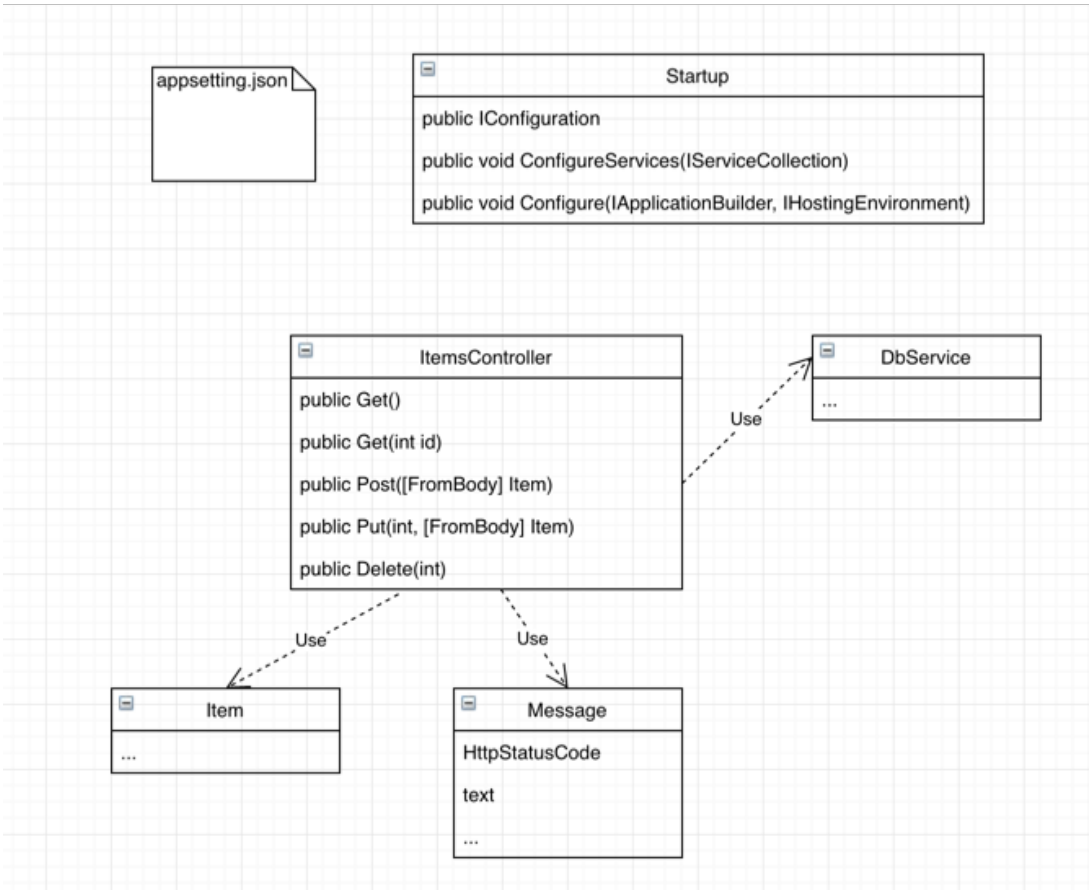

Рисунок 4.4 – Приклад схеми компонента-постачальника

- Конфігураційний файл;
- Файл старту компоннету;
- Опис методів – точок доступу;
- Опис моделей для відповідей.
- Класи взаємодії з ресурсами з данними

Компонент-споживач має мати наступну структуру (рис 4.5):

- Конфігураційний файл;
- Файл старту компоннету;
- Опис методів – точок доступу;
- Опис моделей для відповідей.
- Опис зовнішніх залежностей

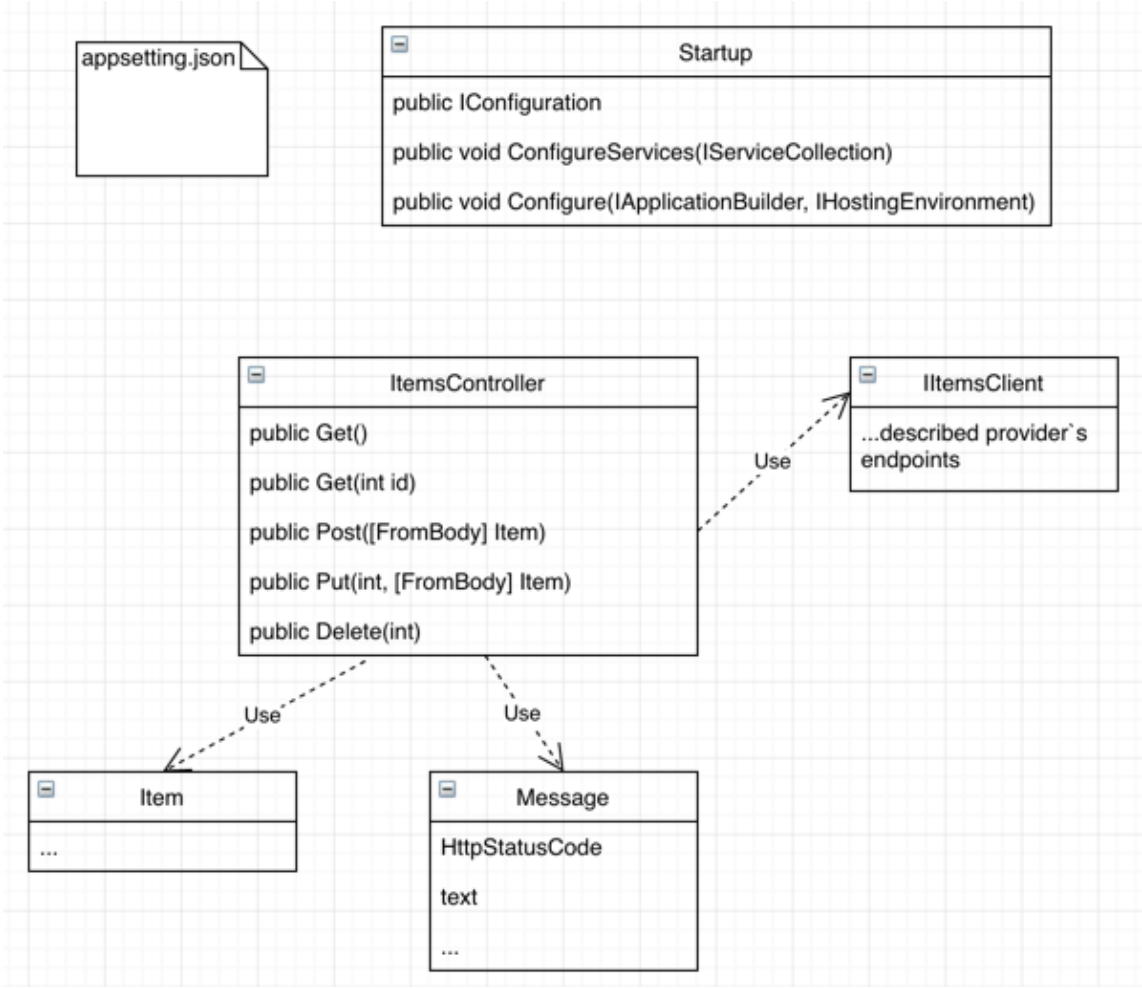

Рисунок 4.5 – Схема компонента-споживача

4.5. Висновки по розділу 4

Для програмної реалізації засобів верифікації інтерфейсів у системах з архітектурою за принципом слабого зв'язування окремих компонентів системи у четвертому розділі виконано наступні завдання:

- визначено формат коду у вигляді бібліотечного фреймворку для реалізації програмного модулю підтримки функціональної верифікації;
- розроблено архітектуру фреймворку написання коду тестів із залученням контактного тестування на платформі .Net Core;
- розроблено та описано прототипи компонентів взаємодії по контракту та емуляторів компонентів;

- сформовано та надано рекомендації з написання API, які планується верифікувати контрактним тестуванням із залученням розробленої бібліотеки фреймворку.

5. Стартап проект

Розділ має на меті проведення маркетингового аналізу стартап проекту для визначення принципової можливості його ринкового впровадження та можливих напрямів реалізації цього впровадження. Проведення маркетингового аналізу передбачає виконання нижченаведених кроків.

5.1 Опис ідеї проекту

В межах підпункту слід проаналізувати та подати у вигляді таблиць:

1. Зміст ідеї (що пропонується).
2. Можливі напрямки застосування.
3. Основні вигоди, що може отримати користувач продукту.
4. Чим відрізняється від існуючих аналогів та замінників.

Перші три пункти подаються у вигляді таблиці (таблиця 5.1) і дають цілісне уявлення про зміст ідеї та можливі базові потенційні ринки.

Таблиця 5.1. Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Програмний модуль для функціональної верифікації інтерфейсів програмної системи	1. Системи з мікросервісною архітектурою	1. Швидкий і якісний процус розробки
	2.Тестування програмних інтерфейсів	2. Верифікація інтерфейсів
	3.Тестування контрактів	3.Не залежить від таргетингової платформи

5.2 Технологічний аудит ідеї проекту

В межах даного підрозділу необхідно провести аудит технології, за допомогою якої можна реалізувати ідею проекту. Визначення технологічної здійсненності ідеї проекту передбачає аналіз таких складових (таблиця 5.2):

1. За якою технологією буде виготовлено товар згідно ідеї проекту.
2. Чи існують такі технології, чи їх потрібно розробити/доробити.
3. Чи доступні такі технології авторам проекту.

Таблиця 5.2. Технологічна здійсненність ідеї проекту

No п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Інтерфейс користувача	Мова програмування C#	Наявна	Умовна безкоштовно
2	Реалізація створення контрактів	Мова програмування C#	Відсутня	Відсутня
3	Формування сценарію	Мова програмування C#	Відсутня	Відсутня
Висновок: проект реалізувати можливо. Обрана технологія реалізації ідеї проекту: Моделювання функціональної верифікації інтерфейсів програмної системи				

За результатами аналізу таблиці робиться висновок щодо можливості технологічної реалізації проекту: так чи ні, а також технологічного шляху, яким це доцільно зробити (з поміж названих технологій обираються такі, що доступні авторам проекту та є наявними на ринку).

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів.

Спочатку проводиться аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку (таблиця 5.3).

Таблиця 5.3. Попередня характеристика потенційного ринку стартап-проекту

No п/п	**Показники стану ринку (найменування)**	**Характеристика**
1	Динаміка ринку (якісна оцінка)	Зростає
2	Наявність обмежень для входу (вказати характер обмежень)	Немає
3	Специфічні вимоги до стандартизації та сертифікації	Немає
4	Середня норма рентабельності в галузі (або по ринку), %	50 %

Середня норма рентабельності в галузі (або по ринку) порівнюється із банківським відсотком на вкладення. За умови, що останній є вищим, можливо, має сенс вкласти кошти в інший проект.

За результатами аналізу таблиці робиться висновок щодо того, чи є ринок привабливим для входження за попереднім оцінюванням.

Надалі визначаються потенційні групи клієнтів, їх характеристики, та формується орієнтовний перелік вимог до товару для кожної групи (таблиця 5.4).

Таблиця 5.4. Характеристика потенційних клієнтів стартап-проекту

N о п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Адаптація концепції контрактного тестування на проекті	Проекти з вефікації програмних систем	Компанії заключають довготривалі договори, а стартапери віддають перевагу пробному терміну	стабільність роботи; наявність випробувального періоду; наявність документації; підтримка необхідних платформ оптимізований час;

Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають (таблиці 5.5-5.6).

Надалі проводиться аналіз пропозиції: визначаються загальні риси конкуренції на ринку. Аналіз пропозиції необхідно виконати аналізуючи існуючі види конкуренцій.

Таблиця 5.5. Фактори загроз

No п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Підходить для нових проектів	Потребує визначеної структури бази даних	Імпорт схеми бази даних
2	Програмний продукт обмежений платформою	Невиликий, адже адаптований під новітні технології	Додавання нових функцій за потреби

Таблиця 5.6. Фактори можливостей

No п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Недоліки в існуючих альтернативах	Існуючі альтернативи або працюють повільно, або не є орієнтованими на конкретну предметну область	Модифікація існуючих платформ

5.4 Розроблення ринкової стратегії проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (таблиця 5.7).

За результатами аналізу потенційних груп споживачів (сегментів) автори ідеї обирають цільові групи, для яких вони пропонуватимуть свій товар, та визначають стратегію охоплення ринку.

Таблиця 5.7. Вибір цільових груп потенційних споживачів

No п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Стартапери	Готові	Високий	Висока	Просто

2	Державні установи	Потребують недовгих переговорів	Середній	Середня	Складно
3	Ентерпрайз	Потребують довгих переговорів	Низький	Низька	Дуже складно
Які цільові групи обрано: стартапери					

Для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку (таблиця 5.8).

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів.

Таблиця 5.8. Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
Орієнтація поточної моделі на ринок стартаперів	Стратегія концентрованого маркетингу	Стартапери потребують швидкості розробки, яку надає підтримка декількох платформ даним продуктом	Стратегія спеціалізації (спирається на диференціацію)

–

Перелік ринкових загроз та ринкових можливостей складається на основі аналізу факторів загроз та факторів можливостей маркетингового середовища.

Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту.

Наступним кроком є вибір стратегії конкурентної поведінки (таблиця 5.9).

Таблиця 5.9. Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів	Чи буде компанія копіювати основні характеристики конкурента	Стратегія конкурентної поведінки
Так	Шукати нових споживачів, забирати існуючих у конкурентів		Стратегія заняття конкурентної ніші

5.5 Аналіз ринкових можливостей запуску стартап-проекту

Для цього у таблиці 5.10 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 5.10. Визначення ключових переваг концепції потенційного товару

Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
Пришвидшення оптимальності роботи алгоритму	Побудова оптимального формування сценарію за оптимальний час	Конкуренти або не мають орієнтованості на електроенергетику, або формують сценарії не оптимальним шляхом

– Надалі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання (таблиця 5.19).

– Після формування маркетингової моделі товару слід особливо відмітити – чим саме проект буде захищено від копіювання.

– Захист може бути організовано за рахунок захисту ідеї товару (захист інтелектуальної власності), або ноу-хау, чи комплексне поєднання властивостей і характеристик, закладене на другому та третьому рівнях товару.

– Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар.

– Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення (таблиця 5.11):

1. Проводити збут власними силами або залучати сторонніх посередників (власна або залучена система збуту).
2. Вибір та обґрунтування оптимальної глибини каналу збуту.
3. Вибір та обґрунтування виду посередників.

Таблиця 5.11. Формування системи збуту

No п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Завантаження з біблотечного сховища	Легість в встановленні	Веб-сайт	Проводити збут силами посередника формування сценаріїв

Останньою складової маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування, визначену специфіку поведінки клієнтів.

5.5 Висновки до розділу 5

Розроблений програмний продукт має переваги над існуючими конкурентами та є конкурентноздатним на ринку. Програма має шляхи подальшого розвитку, визначені маркетингові стратегії та шляхи збуту. Основна цільова аудиторія – це системи формування сценаріїв для розвитку електроенергетики.

6. Висновки

З впровадженням нової архітектури слідує розвиток технологій та інструментаріїв для комфортної підтримки та розгортання додатків. На сьогодні API компонентів та програмних систем має велике значення для їх сполучання і сумісного розвитку. Як любий вихідний код інтерфейси також підлягають тестуванню. Отже, можливість верифікувати інтерфейси завчасно та попереджувати помилки у взаємодії компонентів є важливим фактором успішної та вчасної розробки замовленого продукту та його працездатності. Більш того це створює зручну платформу для розвитку та гнучке і безпечне в цілому для системи внесення змін до існуючих продуктів.

Для вирішення задачі верифікації інтерфейсів у системах, архітектура котрих побудована на принципі слабого зв'язування окремих компонентів системи, в магістерській роботі виконано аналіз існуючих шляхів та методів тестування, які можливо застосувати у верифікації та доведено необхідність формування нового підходу до тестування API та розробки засобів підтримки виконання верифікації інтерфейсів слабо зв'язаних програмних компонентів за запропонованим підходом.

Схема застосування тестувань різних типів для випадку мікросервісів включає шар тестування контрактів (на відміну від її класичного варіанту), тому вирішено побудувати підхід до тестування API при верифікації інтерфейсів на основі саме контрактного тестування.

Для розробки інструменту підтримки застосування контрактного тестування для верифікації інтерфейсів визначено формат коду бібліотечного фреймворку, розроблено архітектуру фреймворку написання коду тестів із залученням контактного тестування на платформі .Net Core, розроблено та описано прототипи компонентів взаємодії по контракту та емуляторів компонентів, а також сформовано та надано рекомендації з написання API, які планується верифікувати контрактним тестуванням із залученням розробленої бібліотеки фреймворку.

Розроблений програмний модуль фреймворку є легким у впровадженні так як програміст-користувач використовує його код як каркас для побудови

власних тестів інтерфейсів. Застосування абстракції високого рівня при конструюванні класів модулю дозволяє легко підлаштувати його до потреб проекту програміста-користувача фреймворку та предметної області.

Модуль легкий у підтримці та ефективний за своїм функціоналом, адже реалізовує первинну взаємодію класичного підходу та найбільш популярних методів.

Список використаних джерел:

1. .Net Core About [Електронний ресурс] – Режим доступу до ресурсу: 1. https://docs.microsoft.com/en-us/dotnet/core/about.

2. Romanov A. Microservices Automation Approach [Електронний ресурс] / Romanov – Режим доступу до ресурсу: 2. https://alexromanov.github.io/2018/09/10/microservices-automation-approach/.

3. Fowler M. Microservices [Електронний ресурс] / Martin Fowler – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html.

4. Microsoft Visual Studio [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/about.

5. Тестування. Теорія [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/forums/topic/13389/.

6. Архітектура REST [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/post/38730/.

7. What is nuget? [Електронний ресурс] – Режим доступу до ресурсу: 4. https://docs.microsoft.com/en-us/nuget/what-is-nuget.

8. Nuget [Електронний ресурс] – Режим доступу до ресурсу: https://www.nuget.org/.

9. HTTP Overview [Електронний ресурс] – Режим доступу до ресурсу: 2. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

10. Introduction to APIs [Електронний ресурс] – Режим доступу до ресурсу: 3. https://zapier.com/learn/apis/chapter-1-introduction-to-apis/#recap.

11. Pact [Електронний ресурс] – Режим доступу до ресурсу: https://docs.pact.io/.

12. Melvin E. Conway. How Do Committees Invent? [Електронний ресурс] / Melvin E. Conway – Режим доступу до ресурсу: http://www.melconway.com/Home/Committees_Paper.html.

13. Create and publich a packege using the dotnet cli [Електронний ресурс] – Режим доступу до ресурсу: 1. https://docs.microsoft.com/en-us/nuget/quickstart/create-and-publish-a-package-using-the-dotnet-cli.

14. Pact-Net [Електронний ресурс] – Режим доступу до ресурсу: 2. https://github.com/pact-foundation/pact-net.

15. Martin Fowler. Tolerant Reader [Електронний ресурс] / Martin Fowler – Режим доступу до ресурсу: https://martinfowler.com/bliki/TolerantReader.html.

16. Martin Fowler. Consumer Driven Contracts [Електронний ресурс] / Martin Fowler – Режим доступу до ресурсу: https://martinfowler.com/articles/consumerDrivenContracts.html.

17. You Arent Gonna Need It [Електронний ресурс] – Режим доступу до ресурсу: http://wiki.c2.com/?YouArentGonnaNeedIt.

18. Martin Fowler. Event Collaboration [Електронний ресурс] / Martin Fowler – Режим доступу до ресурсу: https://martinfowler.com/eaaDev/EventCollaboration.html.

ДОДАТОК А

Акт впровадження

Моделювання функціональної верифікації інтерфейсу програмної системи

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТМ31128_18МП

Аркушів 2

2018

ДОДАТОК А

Акт впровадження

Моделювання функціональної верифікації інтерфейсу програмної системи

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТМ31128_18МП

Аркушів 2

2018